VERIFICATION AND VALIDATION IN SYSTEMS ENGINEERING: APPLICATION TO UML 2.0 ACTIVITY AND CLASS DIAGRAMS

LU'AY ALAWNEH

A THESIS
IN
THE DEPARTMENT
OF
ELECTRICAL AND COMPUTER ENGINEERING

PRESENTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF MASTER OF APPLIED SCIENCE
CONCORDIA UNIVERSITY
MONTRÉAL, QUÉBEC, CANADA

NOVEMBER 2006

Your file Votre référence
ISBN: 978-0-494-28910-5
Our file Notre référence
ISBN: 978-0-494-28910-5

Abstract

Verification and Validation in Systems Engineering: Application to UML 2.0 Activity and Class Diagrams

Lu'ay Alawneh

The increasing complexity of industrial systems requires more efforts to be invested in the process of system verification and validation. The quality of such systems depends on the the different types of techniques that are used to verify and ensure their correct functionality.

The cost of maintaining systems in the latter phases of development is usually very high and may lead in most of the cases to inefficient solutions. Therefore, checking the correctness and validity of systems early in the design phase is greatly desirable. Different verification and validation techniques such as those involving testing and simulation are helpful and useful but may lack in many cases the desired level of rigor and completeness. Moreover, these conventional techniques are generally costly, laborious an time consuming. Conversely, using formal techniques, such as model-checking and program analysis along with design metrics complementary to the conventional verification techniques provides an elevated level of confidence since they are based on theoretical foundations.

Systems Engineering is an interdisciplinary approach that aims to enable the successful realization and deployment of complex systems. Many modeling languages emerged in the systems engineering arena in order to provide the means for capturing and modeling of system's specifications and requirements. The most prominent languages are Unified Modeling Language (UML) 2.0 and Systems Modeling Languages (SysML). Formal verification and software engineering techniques can be applied in order to assess the correctness of different diagrams belonging to the aforementioned modeling languages.

This research work presents a unified paradigm for the verification and validation of software and systems engineering design models expressed in UML 2.0 or SysML. The proposed paradigm relies on an established synergy between three salient approaches, which are model-checking, program analysis, and software engineering techniques.

Acknowledgments

I would like to express my sincere gratitude to my thesis supervisors professor Mourad Debbabi and professor Chadi Assi at Concordia Institute for Information Systems Engineering. Their constructive scientific and technical advice, financial support, hi-tech laboratories and constant guidance had a major influence on the success of the thesis.

I would like also to express my sincere gratitude to my colleagues namely Andrei Soeanu and Yosr Jarraya for the assistance and participation in the research work and experimentation.

Last but certainly not least, I would like to dedicate my thesis to my beloved parents and the rest of my family members for their constant moral support and encouragement which were invaluable in completing this thesis. My thesis is especially dedicated to my precious wife Yasmin for her love and support.

Contents

List of Figures

List of Tables

Chapter 1

Motivations and Background

Modern modeling languages for software and systems, including the most prominent ones, namely UML 2.0 [48] and SysML 1.0a [23, 52] emerged in order to cope with the continuous advancement in software and systems design. The aforementioned modeling languages are playing an increasingly important role in software and systems engineering. Software engineering [30] can be defined as the application of a systematic approach to the development, operation, and maintenance of software, while systems engineering [16] can be said to represent an interdisciplinary approach that enables the realization of successful systems focusing on the system as a whole.

Ubiquitous systems such as hi-tech portable electronics, mobile devices, ATMs as well as many other advanced technologies like aero-space, defense or telecommunication platforms represent important application fields of systems engineering. However, nowadays the critical aspect of software and systems design is not represented by conceptual difficulties or technical shortcomings but it is rather represented by the increased difficulty of assuring bug-free designs. To that effect, the process of design and development mandates a strong, sound, and cost-effective verification and validation (V&V) phase.

Verification is the process of evaluating a system to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase [30]. Conversely, validation is defined as the process of evaluating a system to determine whether it satisfies the specified requirements [30]. The V&V phase can be a major bottleneck in the development life cycle of any complex software or systems engineering product since it can represent about 50% to 80% of the total design effort [5]. Additionally, many engineering solutions are required to meet a very high-level of reliability, security, and performance especially in safety-critical areas. Therefore, ensuring that their predefined requirements are met and that they perform as expected are challenging issues. However, in many modern engineering disciplines, conventional V&V methods such as those involving testing and simulation have become less useful and are not always applicable. Conversely, using formal techniques complementary to simulation provides a certain level of confidence since they are rigorous and complete.

This thesis is part of the research initiative[1] that is supported by the Collaborative Capability Definition, Engineering and Management (CapDEM) project which is an R&D initiative within the Canadian Department of National Defence. The latter aims to the development of a Systems-of-Systems engineering process and relies heavily on modeling and simulation. The aim of this project is to implement a unified approach for V&V in the software and systems engineering fields.

Our approach for V&V in software and systems engineering is based on an established synergy between three major techniques: Formal verification, program analysis and software engineering

[1]This research is the result of a fruitful collaboration between the Computer Security Laboratory (CSL) at Concordia University and Defence Research and Development Canada (DRDC) at Ottawa. The research is supported by the CapDEM (Collaborative Capability Definition, Engineering and Management) project.

techniques. By formal verification, we imply model-checking. By program analysis, we mean data and control flow analysis. By software engineering techniques, we mean metrics which are used to measure quality attributes of object-oriented design.

The benefits of the proposed approach are manifold. Our approach inherits rigor and soundness from the use of formal techniques. Moreover, it is cost-effective since it is applied in the early stages of the development process. In fact, early and efficient identification of flaws in the design can have economical advantages if compared to the same task done in the maintenance phase [9]. Furthermore, it can be entirely automatic thus requiring no related background for systems engineers. In addition, different qualitative and quantitative attributes can be measured using software engineering metrics in order to assess the overall design quality. Also, to the best of our knowledge, this is a pioneering endeavor in using these three techniques together. Moreover, the results of our research initiative have been published in several international conferences [2–4].

This thesis focuses on the software engineering techniques applied to UML 2.0 class and package diagrams and formal verification methods applied to UML 2.0 activity diagrams.

In the following section, we describe our approach and the underlying techniques that are utilized.

1.1 Approach

As previously stated, the foundation of our approach lies in the harmonious synergy between three well established techniques that are: Formal analysis (model-checking), software engineering techniques (metrics), and program analysis (static analysis). Figure 1.1 depicts the synoptic overview of our approach. In the following paragraphs, we describe the three distinctive layers sustaining our approach. First, we explain the motivation of our choice. Then, we present how we intend to use them in the context of V&V of design models and finally we give a description of our framework.

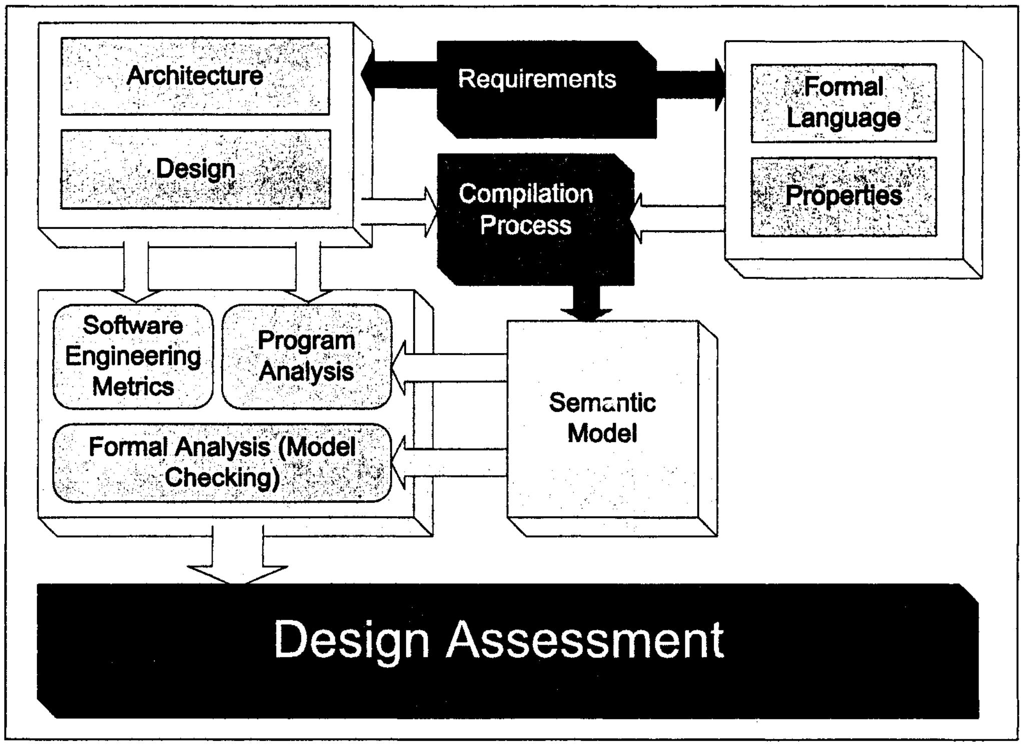


Figure 1.1: Synoptic Overview of the V&V Process

First, model-checking is a model-based verification technique. It is fully automated and it has been used for the verification of real applications, both software and hardware systems, including digital circuits, controllers, and communication protocols. SPIN [27] and SMV [32] are examples of model checkers. We chose to work with NuSMV [15], a modified version of the SMV, since it supports fairness constraints along with branching-time logic for property specification, namely the Computation Tree Logic (CTL). The latter has an interesting expressiveness that allows to specify many useful properties such as those related to deadlock, reachability and state sequencing to name just a few. Model-checking is meant to achieve the dynamic assessment of the model. Thus, it is used to check whether the dynamic aspect of a model satisfies the specified properties (e.g. safety or liveness). Accordingly, in order to use it, one has to extract the semantics model from the behavioral diagram to be verified (e.g. activity diagram). Model-checking has been successfully used in medium-sized complex designs. Even though this technique was generally coupled with severe scalability issues, numerous efforts tackle this problem in various ways, such as on-the-fly model-checking [53], symbolic model-checking [31], and distributed on-the-fly symbolic model-checking [8].

The second layer is represented by program analysis techniques such as flow analysis. We advocate their use in verifying important model properties such as data dependencies, control dependencies, invariants, anomalous behavior, reliability, and compliance to certain specifications.

Finally, the third layer consists of a set of fifteen metrics that we have adopted from software engineering field [1, 2, 14, 37]. We advocate their use to assess quality attributes of various models independently of their underlying discipline (software or systems engineering). More precisely, this feature enables us to assess the structural aspects of the systems model. We found in the literature some support about the use of metrics in systems engineering. For instance, Tugwell et al. [60] outline the importance of metrics in systems engineering especially related to complexity measurement.

In addition to applying metrics on the structural diagrams such as class diagram, they can also be applied on the semantics model derived from different behavioral diagrams. For example, cyclomatic complexity and length of critical path could be applied on the semantics model. Thus, the quality assessment of a given design can combine both the static and dynamic perspectives. The aim is to be able to compare the structural and the behavioral views qualitatively and quantitatively. For example, comparing the complexity of the activity diagram and the one of its corresponding semantics model can contrast how close is the diagram structure reflecting the behavior. If the complexity of the semantics model is less than the one for the original diagram, this can imply that some parts of the structure are not used or redundant or meaningless.

1.1.1 V&V Framework

Our V&V framework requires an underlying modeling tool wherefrom various models can be fetched and assessed. Our choice is ARTiSAN Real-time Studio which is a modeling tool that supports UML and SysML designs. Additionally, it provides component-based development specifically for real-time systems [58, 59].

The current version of our framework is composed of three core components, as shown in Figure 1.2. First, we have the semantic compilation component responsible for deriving the semantic model of a specific diagram. It communicates with the model checker by providing the semantic model along with the properties to be verified. Second, we have the metric computation component that is used for applying metric algorithms. We have provided an interface that accesses the object repository of the modeling tool and retrieves the needed information about the diagrams. Finally, the assessment results component is devoted to the presentation of interpreted results. With respect to the dynamic assessment, whenever a specified property fails, the trace provided by the model-checker is analyzed and the relevant information is provided as feed-back.

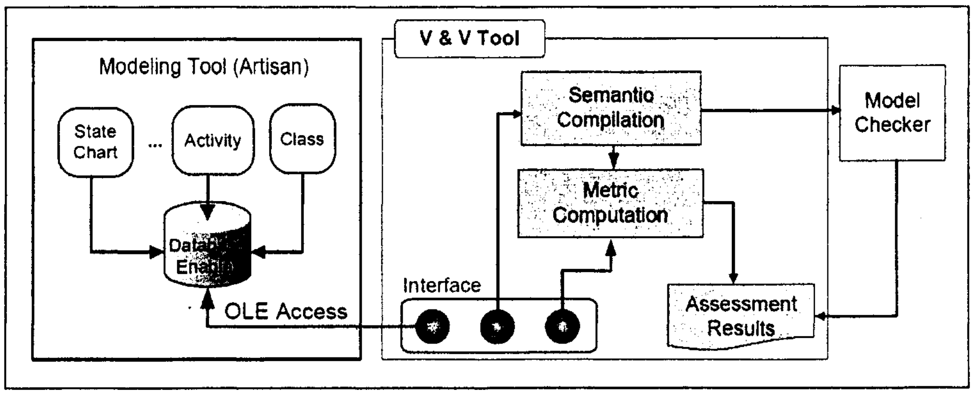


Figure 1.2: Architecture of the Framework

Our approach targets a number of important system aspects that can be captured using different UML diagrams such as state machine, activity, sequence, class and package diagrams. Subsequently, we briefly present some related definitions for the selected set of system aspects.

1.1.2 System Aspects

There are many systems engineering aspects including requirements, structure, concurrency, and performance. In the sequel, we briefly present those that we target in this work:

- *Requirements*: They are a description of what a system should do and are captured by requirement diagrams in SysML or using sequence and use case diagrams in UML 2.0.
- *Time*: It is captured by timing diagrams, which provide a visual representation of objects changing state and interacting over time.
- *Concurrency*: It identifies how activities, events, and processes are composed (sequence,

branching, alternative, parallel, etc.). Concurrency could be specified using sequence, activity, and timing diagrams.

- *Performance*: It is the total effectiveness of the system. It makes reference to the timeliness aspects of how systems behave. This includes different types of quality of service characteristics such as latency and throughput. Timing and sequence diagrams depict performance aspects.

- *Structure*: It is shown in class and composite structure diagrams. The class diagram shows the relationships between different classes of the system. The composite structure diagram shows the internal structure of the building blocks of the system and how these blocks are interfacing with other components of the system.

- *Interface*: It identifies the shared boundaries of the different components of the system whereby the information is passed. This aspect is shown using class diagrams in UML 2.0 and SysML, composite structure diagrams in UML 2.0, and assembly diagrams in SysML.

- *Control*: It identifies the order in which actions, states, events, and/or processes are arranged. It is captured using state machine, activity, and sequence diagrams in UML 2.0 and SysML.

Though there are other system aspects, the most relevant ones to the verification process of UML diagrams have been enumerated in this section.

1.2 Structure of the Thesis

The remaining of this thesis is structured as follows: in Chapter 2, we present the state-of-the-art in the verification and validation research initiatives targeting UML structural and behavioral diagrams.

Chapter 3 outlines the historical context and background that led to the emergence of the UML 2.0 and SysML modeling languages. Moreover, it gives a brief description of the several diagrams in the targeted modeling languages.

In Chapter 4, we introduce the UML 2.0 activity diagram along with the model checker used for its verification and validation. Furthermore, we explain our algorithm that is used to generate the required transition system for the verification process. The verification and validation tool architecture is explained along with an example that illustrates our methodology when applied to the verification of workflow systems modeled using activity diagrams. We conclude this chapter with a discussion about the model-checking feasibility and the related open problems.

Chapter 5 enumerates system quality attributes that can be assessed when applying software techniques on UML class and package diagrams. Thereafter, a set of fifteen software metrics is presented along with the explanation thereof. Each metric is presented separately with the formula for its calculation and its corresponding nominal range. Moreover, a set of three proposed metrics for assessing the functionality of class diagrams are explained. The chapter concludes with a class and package diagram example along with the analysis results.

Finally, conclusions and future work are presented in Chapter 6.

Chapter 2

Related Work

This chapter surveys the state of the art in terms of V&V of software and systems engineering design models. Particularly, we focus on the verification of UML 2.0 and SysML design models. In the following, we present V&V of the most prominent UML diagrams namely class and package, state machine, activity, and sequence diagrams.

2.1 Software Engineering Techniques

Complexity of software systems is increasing dramatically which encourages the need for some techniques to assure better system quality. System quality should be controlled in the early stages of design. A good software system offers components that are more robust, more maintainable, more reusable, etc. In the literature, many object oriented metrics have appeared to bring up highly reliable software systems. Software metrics are efficient methods for assessing the quality of UML class and package diagrams since they give an insight about the complexity and structure of software systems. In the following, we present a list of contributions in the field of software engineering techniques and proposals for several metrics suites for UML class and package diagrams.

2.1.1 Chidamber and Kemerer Metrics Suite

Chidamber and Kemerer [14] proposed a set of six metrics for object oriented designs. This metrics suite aims to measure the diagram's complexity by applying the metrics on different quality attributes such as maintainability, reusability, etc. From these six metrics, only three can be applied on UML class diagrams. In the following, we present these metrics that are related to UML class diagrams:

- Coupling Between Object Classes (CBO). This metric measures the level of coupling among the classes in the diagram. A class that is excessively coupled to other classes in the diagram is disadvantageous to modular design and prohibits reuse and maintainability.
- Depth of Inheritance Tree (DIT). This metric represents the length of inheritance tree from a class to its root class. A deep class in the tree inherits a relatively high number of methods which in turn increases its complexity.
- Weighted Methods per Class (WMC). It is the summation of the complexity of all methods in the class. A simpler case for WMC is when the complexity of each method is considered as unity. In this case, WMC is considered as the number of methods in the class. A high WMC value is a sign of high complexity and less reusability.

2.1.2 MOOD Metrics Suite

Abreu et al. [13] proposed a set of metrics to assess the structural mechanisms of the object-oriented paradigm such as enacapsulation, inheritance, polymorphism. The MOOD (Metrics for Object-Oriented Design) metrics suite can be applied on UML class diagrams. In the following, we present these metrics that are relevant to our field of study:

- Method Hiding Factor (MHF). This metric is a measure of the encapsulation in the class. It is the ratio of the sum of hidden methods (private and protected) to the total number of methods defined in each class (public, private and protected). If all the methods in the class are hidden then the value of MHF is high and indicates that this class is not accessible and thus not reusable. If the value of MHF is 0, this assumes that all the methods of the class are public which hinders encapsulation.

- Attribute Hiding Factor (AHF). This metric is likewise the average of the invisibility of attributes in the class diagram. It is the ratio of the sum of hidden attributes (private and protected) for all the classes to the sum of all defined attributes (public, private and protected). A high AHF value indicates appropriate data hiding.

- Method Inheritance Factor (MIF) and Attribute Inheritance Factor (AIF). These two metrics are a measure of the class inheritance degree. MIF is calculated as the ratio of all inherited methods in the class diagram to total number of methods (defined and inherited) in the diagram. AIF is calculated as the ratio of all inherited attributes in the class diagram to the total number attributes (defined and inherited) in the diagram. A zero value indicates no inheritance usage which may be a flaw unless the class is a base class in the hierarchy.

- Polymorphism Factor (POF). This metric is a measure of methods overriding in a class diagram. It is the ratio between the number of overridden methods in a class and the maximum number of methods that can be overridden in the class.

- Coupling Factor (COF). COF measures the coupling level in the class diagram. It is the ratio between the actual couplings among all classes and the maximum number of possible couplings among all the classes in the diagram. A class is coupled to another class if methods of the former access members of the latter. High values of COF indicate tight coupling which increases the complexity and hinders its maintainability and reusability.

2.1.3 Li and Henry's Metrics Suite

Li and Henry [38] proposed a metrics suite to measure several class diagram internal quality attributes such as coupling, complexity and size. In the following, we present the main two metrics proposed by Li and Henry that can be applied on UML class diagrams:

- Data Abstraction Coupling (DAC). This metric calculates the number of attributes in a class that represent other class types (composition). This metric measures the coupling complexity due to the existence of abstract data types (ADT). The complexity due to coupling increases if more ADTs are defined within a class.

- SIZE2. This metric is defined as the number of attributes and the number of local methods defined in a class. This metric is a measure of the class diagram size.

2.1.4 Lorenz and Kidd's Metrics Suite

Lorenz and Kidd [40] proposed a set of metrics that measures the static characteristics of software design such as size, inheritance and the internal attributes of the class. The first size metric is the public instance methods (PIM). This metric is the count of public methods in a class. The second metric is the number of instance methods (NIM) and is the count of all methods (public, protected and private) in a class. The last metric, number of instance variables (NIV) counts the total number of variables in a class.

Furthermore, they proposed another set of metrics that measures the class inheritance usage degree. Herein, we give a glimpse of these metrics and in Chapter 5 we explain them in detail. The NMO metric gives a measure of the number of methods overridden by a subclass. The NMI is the total number of methods inherited by a subclass. Additionally, the NMA metric is the count of the methods added in a subclass. Finally, the NMO and DIT [14] metrics are used to calculate the specialization index (SIX) of a class, which gives an indication of the class inheritance utilization.

2.1.5 Robert Martin Metrics Suite

Robert Martin [42] proposed a set of three metrics that is applicable for UML package diagrams. This set of metrics measures the interdependencies among packages. Packages that are highly interdependent tend to be not flexible which accordingly states that those packages or subsystems are hardly reusable and maintainable. Therefore, interdependency among packages in a system should be taken into consideration. The three metrics defined by Robert Martin are *Instability, Abstractness* and *Distance from Main Sequence (DMS)* respectively. The Instability metric measures the level of instability of a package. A package is unstable if it depends more on other packages than they depend on it. The Abstractness metric is a measure of the package's abstraction level which depends on its stability level. Finally, the DMS metric measures the balance between the abstraction and instability of a package. In Chapter 5, we discuss these three metrics and explain their usefulness in detail.

2.1.6 Bansiya et al. Metrics Suite

Bansiya et al. [6] defined a set of five metrics to measure several object oriented design properties such as data hiding, coupling, cohesion, composition and inheritance. In the following, we present only those metrics that can be applied to UML class diagrams. The data access metric (DAM) measures the level of data hiding in the class. DAM is the ratio of the private and protected (hidden) attributes to the total number of defined attributes in the class. The direct class coupling metric (DCC) is the count of the total number of classes that a class is coupled. The measure of aggregation (MOA) computes the number of attributes defined in a class whose types represent other classes (composition) in the model. In their work, Bensiya et al. applied their metrics suite to some case studies and defined nominal ranges for their metrics based on their observations.

2.1.7 Briand et al. Metrics Suite

Briand et al. [12] proposed a metrics suite to measure the coupling among classes in the class diagram. These metrics determine each type of coupling and the impact of each type of relationship on the class diagram quality. In their work, Briand et al covered almost all types of coupling occurrences in a class diagram. These types of relationships include coupling to ancestor and descendent classes, composition, class-method interactions, and import and export coupling. Consequently, as a result of their work, they applied their metrics suite on two real case studies and observed that coupling is an important structural aspect to be considered when building quality models of object-oriented design. Moreover, they also concluded that import coupling has more impact on fault-proneness than export coupling.

2.2 Formal Verification Methods using Model-checking

Formal verification refers to various scientific and engineering techniques for verifying and validating the correctness of systems. These techniques are often based on mathematical logic and can be used to perform verification and validation on a number of UML and SysML diagrams. There are two major formal verification techniques: theorem-proving [34] and model-checking [55]. These methods can reveal inconsistencies, ambiguities, incompleteness as well as several other shortcomings in a system. On the other hand, previous verification methods such as simulation [44] and testing [61] cover only a subset of system properties. In contrast, model-checking provides precise and exhaustive verification for the targeted properties that the system must satisfy. In the following, we present a number of approaches for the verification and validation of UML behavioral diagrams that use several model-checking techniques.

2.2.1 Tool Support for Verifying UML Activity Diagrams

Rik Eshuis and Roel Wieringa [21] developed a tool for the verification of workflow models specified in UML activity diagrams. The tool reads the UML activity diagram and converts it to its corresponding activity hypergraph. In the latter, all the pseudo states (Branch, Merge, Fork, and Join) are replaced with their equivalent hyperedges which are edges with multiple source and/or target states. This hypergraph is then applied to a specific algorithm that will generate a transition system that covers all the configurations that a system might abide in. These properties to be checked on this transition system are expressed as CTL formulas and verified by the NuSMV model checker.

2.2.2 Framework for the Verification of UML Models

Esther Guerra and Juan de Lara [25] proposed a framework for the verification of UML models by building meta-models for UML diagrams and then translating them into formalisms that enable to check their properties. They describe the translation (denotational semantics) as well as the formalisms operational semantics by means of graph grammars. They also implemented the AToM3 multi-paradigm tool that translates UML diagrams into petri-nets notation for subsequent verification using model-checking.

2.2.3 Tool for Verifying UML Models: vUML

Johan Lilius et al. [39] developed the *vUML* for automatic verification of UML state-chart diagrams. The main purpose of vUML is to verify concurrent and distributed models containing active objects. Furthermore, vUML can be used to verify sequential designs since it supports synchronous communication. The tool uses SPIN model checker to perform the verification on the corresponding PROMELA code for state-chart diagrams. If an error is found during the verification, the tool creates a UML sequence diagram that shows how to reproduce the error in the model.

2.2.4 Towards a Formal Operational Semantics of UML State-chart Diagrams

In their work, D. Latella et al. [36] presented a formal semantics for UML state-chart diagrams based on Kripke structure. The first step in the verification process is to map the state-chart to the intermediate format of Extended Hierarchical Automata (EHA). Finally, the EHA structure is specified using PROMELA code. The latter is input to the SPIN model checker for verification.

2.2.5 Model-Based Verification and Validation of Properties

In [20], the authors show how model-based property analysis can be made applicable within a UML-based development process. Examples for such properties include deadlock freedom, timing consistency, and limited memory resources. The approach is to design a partial formalization of UML models such that existing verification techniques can be reused. The method employed is based on graph transformation techniques to automate the translation of UML models into CSP [26] which is a formal language accepted by the FDR [41] model checker.

2.2.6 Hugo/RT

Hugo/RT [29] is a UML model translator for model-checking and theorem proving. The main focus of the model-checking component of Hugo is to verify the consistency of UML state machines against specifications expressed as collaboration or sequence diagrams [33]. Hugo/RT translates a subset of UML models, namely state machines, collaborations, interactions, and OCL constraints to timed automata. Moreover, it implements two toolkits that translate the timed automata to a suitable input to the UPPAAL [18] real-time model checker and to the SPIN [27] on-the-fly model checker respectively. Additionally, it can translate the UML models into the system language used by the KIV theorem prover [33].

2.3 Summary

This chapter presented the state-of-the-art in the verification and validation of UML software and systems engineering design models. Our approach targets a set of software metrics that were presented in this chapter. Moreover, our approach is inspired from the work of Rik Eshuis et al. [21] for using the NuSMV model-checker in the verification and validation of UML 2.0 activity diagrams.

Chapter 3

Systems Engineering Modeling Languages

3.1 Introduction

A system can be viewed as a set of interrelated components that interact with one another in an organized fashion towards a common purpose. The components of a system may be quite diverse, consisting of persons, organizations, procedures, software, equipment, etc. In order to behave correctly, a system must satisfy its design requirements. Thus, the need for an interdisciplinary approach and means to enable the realization of successful systems stimulated the establishment of the systems engineering discipline. Systems engineering [16] is a promising discipline that aims to check if the system is designed, built, and operated in the most cost-effective way, taking into consideration its performance, maintainability, safety, etc. It is an interdisciplinary approach encompassing the entire technical effort to evolve and verify system quality through its development, manufacturing, verification and deployment. This chapter outlines the historical context and background that led to the emergence of UML 2.0 and SysML.

3.2 Systems Engineering Standardization

Systems engineering should be internationally standardized to facilitate the collaboration among systems engineers. Moreover, it will lead to compatible systems engineering technologies. Therefore, many organizations have been working on providing standard frameworks and modeling languages for systems engineering. In the following sections, we present some of the main organizations that are dedicated to standardize the systems engineering discipline such as the Object Management Group (OMG) [22], INternational Council On System Engineering (INCOSE) [16], and the International Standard Organisation (ISO) [51]. Also, we present some initiatives and modeling languages related to systems engineering developed by these organizations.

3.2.1 Object Management Group

The Object Management Group (OMG) [22] is an international association founded in 1989 that aims to provide standards for object-oriented systems to help in reducing the complexity and costs and hastening the introduction of new software technologies. OMG has introduced a new suite of specifications that will accomplish the aforementioned goals. This suite of specifications will lead to interoperable, reusable, and portable software systems components as well as data models based on standardized models. Many standards were established and developed by OMG that became widely used in the industry. Subsequently, we present those standards that are related to systems engineering.

Model Driven Architecture

The Model Driven Architecture (MDA) [46] is based on the idea of separating the specifications of a system from its implementation in a designated platform. Therefore, MDA focuses on modeling software systems in a way that allows their functionality and behavior to be separated from their implementation details. The created model which is called the platform independent model (PIM) enables thereafter the application to be easily ported from one environment to another. This PIM is then translated into one or more platform specific models (PSM). Thus, the separation of the model from the implementation allows to cover a broad range of systems development projects such as electronic commerce, financial services, healthcare, aerospace and transportation. The three major goals of MDA are portability, interoperability, and reusability through architectural separation of concerns.

Unified Modeling Language

UML [35] is a modeling language adopted by OMG in 1997 that enables systems developers to specify, visualize, and document software models. These models are abstract representations of the implementation details of systems. The advantage of using an abstract model is to enable designers to prepare the proper mold for the system implementation. UML is described in detail in Section 3.3.

Meta-Object Facility

MOF [47] defines an abstract language and framework for specifying, constructing, and managing technology neutral metamodels. The MOF specification is intended to provide information on modeling capabilities. It is the standard of OMG for defining modeling languages and their interoperability. In this context, the MOF model is referred to as a meta-metamodel since it is being used to define metamodels such as UML. The MOF provides a formal and clear semantics for each element of the UML metamodel.

XML Metadata Interchange

XMI [49] is an OMG standard for exchanging metadata information via Extensible Markup Language (XML). It can be used for any metadata whose metamodel can be expressed in Meta-Object Facility (MOF). The most common use of XMI is as an interchange format for UML models, although it can also be used for serialization of models described in other languages (metamodels).

3.2.2 INternational COuncil on Systems Engineering

INCOSE [16] is an international professional society for systems engineers whose mission is to foster the definition, understanding, and practice of systems engineering in industry, academia, and government. INCOSE was formed in 1992 to develop and enhance the interdisciplinary approach to enable the realization of successful systems. Furthermore, INCOSE was a major player in the adoption of the new modeling language for systems engineering, namely SysML.

Systems Modeling Language

SysML [52] defines a general-purpose modeling language for systems engineering applications. It supports the specification, analysis, design, verification, and validation of a broad range of complex systems that may include hardware, software, data, personnel, procedures, and facilities. SysML inherits and extends the features and characteristics of UML focusing on the systems engineering aspects. Section 3.4 explains SysML in more detail.

3.2.3 International Organization for Standardization

ISO [51] is the well-known world's largest developer of standards. Those standards are useful to industrial and business organizations, governments and other regulatory bodies. ISO 10303 is one of the standards related to product data representation and exchange, which is implemented in XML. Also, ISO 10303 is introduced as STEP (STandard for the Exchange of Product).

Standard for the Exchange of Product

STEP is an ISO project meant to develop mechanisms for the representation and exchange of digital product data in a neutral form. The term product data denotes the totality of data elements that completely define a product for all applications over its expected life cycle. The goal of the project is to enable a product representation to be exchanged without any loss of completeness or integrity.

AP233

AP233 is a data exchange protocol for systems engineering data, based on ISO 10303 and represents a neutral data exchange schema for systems engineering data. AP233 is used to support the whole system development life cycle ranging from requirements definition to system verification and validation.

3.3 Unified Modeling Language

The Unified Modeling Language (UML) [22, 54, 56] enables system developers to specify, visualize, construct, and document the artifacts of software systems. These models are used to define an abstract module for the implementation details of software system and systems in general. The abstract overview of the system helps developers obtain an accurate system design before actual implementation. The latest UML version is UML 2.0, it was developed to overcome the shortcomings of the earlier UML 1.x versions. UML 2.0 has an increased level of precision in describing the basic modeling concepts and their semantics. Moreover, it has an improved capability to model large-scale software systems. This includes the ability to model entire system architectures to use UML as an architectural description language. Therefore, UML 2.0 is a suitable modeling language for systems engineers to model their systems and apply verification and validation on their models. In the following, we describe the evolution of UML and give a brief description of the existing UML 2.0 diagrams.

3.3.1 UML History

In the early 1990s, several object-oriented modeling languages were developed in the software engineering community. Nevertheless, these modeling languages were not satisfactory for the software community. Therefore, the need for a united solution was crucial. The idea behind UML emerged in late 1994 by Grady Booch [10] from Rational Software Corporation to bring up a rich modeling language. This modeling language was designed specifically to represent object-oriented systems based on Booch's methodology and the Object Modeling Technique (OMT) by

Rumbaugh [50]. The merge of the aforementioned methods in the fall of 1995 emerged with the birth of the Unified Modeling Language (UML 0.8). In June 1996, UML 0.9 was released with the merge of Ivar Jacobson's Object-Oriented Software Engineering (OOSE) [56] method. The standard has since progressed through versions 1.1 and 1.3 on to version 1.4. Although UML 1.x was widely accepted in the community, it had some shortcomings such as the lack of support for diagram interchange. Also, it is considered as being too complex and having an inadequate semantics definition. Moreover, UML 1.x is not fully aligned with MOF and MDA. Thus, a major revision was required to address these problems [19]. Consequently, UML 2.0 is the new adopted official version by OMG.

3.3.2 UML 2.0 Diagrams

UML 2.0 diagrams are classified into two categories: structural, and behavioral diagrams. The latter includes a subset known as interaction diagrams. Figure 3.1 depicts the UML 2.0 diagrams taxonomy. In the following, we present each category with its corresponding diagrams.

Structural Diagrams

Structural diagrams depict the static features of a model and the relationships and dependencies among the model elements.

The *class diagram* addresses the static design view of a system. It shows the classes that construct the system and the relationships among them. Classes relate to each other through different relationships such as association, aggregation, composition, dependency, and inheritance relationships.

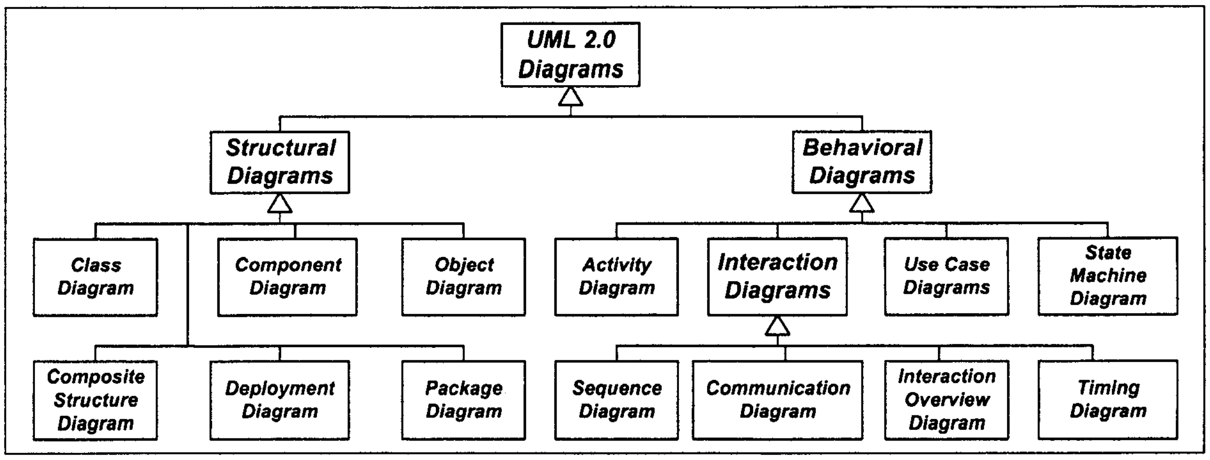


Figure 3.1: UML 2.0 Diagrams Taxonomy

The *component diagram* addresses the static implementation view of a system. It is used to show the organization and dependencies among system's components through well defined interfaces. A component diagram has a higher level of abstraction than a class diagram, usually a component is implemented by one or more classes (or objects) at runtime.

The *composite structure diagram* depicts the internal structure of a classifier, such as a class, a component or a collaboration, including the interaction points (ports) of the classifier to the system. Moreover, a composite structure diagram shows the configuration and relationship of parts that together perform the behavior of the containing classifier.

The *object diagram* shows how specific instances of a class are related to each other at run-time. The object diagram consists of the same elements as its corresponding class diagram. However, the class diagram defines the classes with attributes and methods. Whereas, in the object diagram the class attributes and method parameters are assigned values. Object diagrams aim to support the study of requirements by modeling examples from the problem domain (object diagrams may be used as test cases later). Moreover, they are used to validate class diagrams by modeling test cases.

The *package diagram* organizes and groups the elements and shows dependencies among them.

The *deployment diagram* depicts the hardware architecture of a system and the components that run on each piece of hardware. A deployment diagram is needed for applications that run on several devices since it shows the hardware, the software, and the middleware used to connect the devices of the system. Moreover, deployment diagrams can be used to represent the architecture of embedded systems to show how hardware and software components interact in the system.

Behavioral Diagrams

Behavioral diagrams capture the varieties of interactions and instantaneous states within a model as it executes over time.

The *use case diagram* describes a set of scenarios that show the interaction between the users and a system. A user can be either a person or another system. A use case refers to the sequence of actions that the system can perform by interacting with its actors. Use cases are used in the initial phase of the project. They are useful in defining the requirements of the project and to identify the expectations of a system.

The *activity diagram* depicts the flow of activities within a system. It addresses the dynamic view of a system to describe the procedural logic, business process and workflow. An activity diagram may contain many processing paths that contain decision making and parallel processing. Thus, it is helpful to describe behavior that involves several objects collaborating across several use cases.

The *state machine diagram* captures the dynamic behavior of a system. It describes the significant states that an entity can be during its life-cycle. The state of an entity changes in response to environmental stimuli or events. Therefore, state machine diagrams are useful for modeling of real-time systems and reactive systems.

The *protocol state machine diagram* is a diagram introduced in UML 2.0. It is based on the state machine diagram but focuses on expressing the protocol of operations that shows the pre and post conditions without showing the object behavior.

Interaction Diagrams

Interaction diagrams represent a subset of behavioral diagrams and are used to emphasize the flow of control and data among the blocks in the system being modeled.

The *sequence diagram* shows object interactions arranged in a time sequence. Sequence diagrams identify the communication required to fulfill an interaction. Moreover, they show the objects that participate in an interaction and the messages used to trigger the interactions among the objects.

The *interaction overview diagram* provides a high-level view of the logical progression of the execution of the system through a set of interactions. The interaction overview diagram uses the syntax and semantics of an activity diagram to model the flow of logic in a series of interactions expressed as sequence diagrams.

The *communication diagram*, known as the collaboration diagram in UML 1.x versions, focuses on the structure and the sequence of messages among objects at run-time during a collaboration instance. Generally, it shows instances of classes, their interrelationships, and the message flow among them.

The *timing diagram* is a different way to present a sequence diagram. It explicitly shows changes in the system on a time line. It is a new diagram in UML 2.0. Timing diagrams are used to document the timing requirements that control the changes in the state of the system. Moreover, they can be used with the state machine diagram when the timing of events is more critical for a proper operation of the system.

3.4 Systems Modeling Language

Systems Modeling Language (SysML) [52] is a general purpose modeling language for systems engineering applications. SysML supports the specification, analysis, design, verification and validation of a broad range of complex systems. These systems may include hardware, software, information, processes, personnel, and facilities.

3.4.1 SysML History

The birth of the SysML initiative can be traced to a strategic decision by INCOSE's [16] Model Driven Systems Design workgroup in January 2001 to customize the Unified Modeling Language (UML) for systems engineering applications. As a result, a collaborative effort between INCOSE and OMG [22] arose to jointly charter the OMG Systems Engineering Domain Special Interest Group (SE DSIG) [1] in July 2001. The SE DSIG, with support from INCOSE and the ISO AP 233 workgroup, developed the requirements for the modeling language, which were subsequently issued by the OMG as part of the UML for Systems Engineering Request for Proposal in March 2003. In January 2004, the first SysML draft was submitted to OMG and was also reviewed by

[1] Systems Engineering Domain Special Interest Group, http://syseng.omg.org/

INCOSE. In November 2004, INCOSE submitted its review of the first draft to OMG. Later on, in the first quarter of 2005, SysML was adopted by OMG. Currently, SysML 1.0 specifications was issued by OMG.

3.4.2 SysML diagrams

SysML is developed to customize UML for systems engineering purposes. Therefore, SysML adopts a subset of UML 2.0 specifications and adds new constructs for systems engineering. Figure 3.2 depicts the SysML diagrams taxonomy. This section presents the new diagrams in SysML that do not exist in UML 2.0. Moreover, it explains the main new features and modifications to UML 2.0 customized diagrams. Moreover, few diagrams from UML 2.0 were not considered in the SysML specifications. These diagrams are the object, component, deployment, package, communication, timing, and interaction overview diagrams.

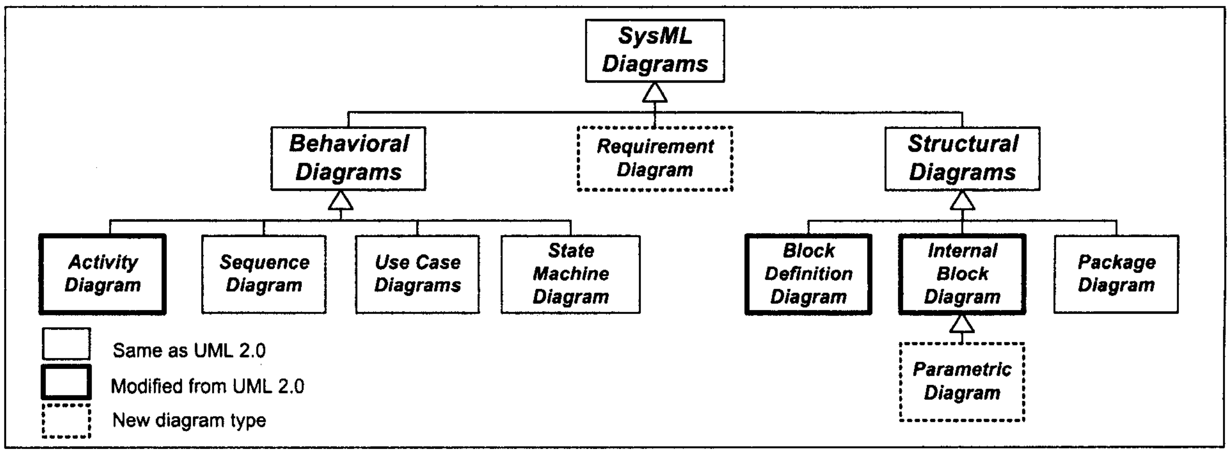


Figure 3.2: SysML Diagrams Taxonomy

New SysML Diagrams

The *requirements diagram* explicitly captures requirements and their relationships. It is intended to assist in integrating SysML models with requirements management tools. This diagram uses dependencies (satisfy, trace, derive, decompose, verify) with stereotypes to detail a relationship. Moreover, it is possible to show traceability between model elements and requirements.

The *parametric diagram* is commonly used to model properties and their relationships. They are used to represent complex mathematical and logical expressions or constraints. The parameter (i.e., temperature or pressure) is a measurable factor that varies in experiments that define a system and determine its behavior. The parametric diagram aims to bring more compliance with other system modeling tools by defining a set of quantifiable characteristics and relationships between them. Moreover, the parametric diagram shows how changing a value of a property impacts the other properties in the system. Therefore, this diagram can be used to perform simulations on models by modifying parameter values and observing the impact on the whole system. Additionally, the parametric diagram is useful in analyzing the performance and reliability of a system by identifying the conditions that could make the system unreliable or malfunctioning.

Modified Diagrams from UML 2.0

The *block definition diagram* defines features of a block and relationships between blocks such as associations, generalizations, and dependencies. It captures the definition of blocks in terms of properties, operations, and relationships such as a system hierarchy or a system classification tree. Moreover, it allows to develop a number of conceptual architectures which can be used as a starting point for trade-off analysis. The block definition diagram is based on the UML class diagram with some restrictions and extensions to its features. Some capabilities available for UML classes, such as more specialized forms of associations, have been excluded from SysML blocks to simplify the language. Furthermore, notational and metamodel support for n-ary associations and qualified associations has been excluded from SysML.

The *internal block diagram* captures the internal structure of a block in terms of properties and connectors between properties. It is based upon the UML composite structure diagram but also excluded many elements. A block can include properties to specify its values, parts, and references to other blocks. Ports are a special class of property used to specify allowable types of interactions between blocks. Constraint Properties are a special class of property used to constrain other properties of blocks.

The *activity diagram* is used to depict control and data flow throughout the system. SysML added new features to UML activity diagram such as continuous flow semantics, activity parameters, action pins. Moreover, it extends edges and output parameters with expression evaluating to constant probabilities. It also provides a model library for Enhanced Functional Flow Block Diagram.

3.5 Summary

This chapter explained the importance of the systems engineering domain and the reasons that led to the emergence of the modeling languages for better systems designs focusing on the two main modeling languages adopted in the software and systems engineering communities.

Chapter 4

V&V of UML 2.0 Activity Diagrams

4.1 Introduction

UML activity diagrams [56] depict system behavior using a control flow and data flow model and can be typically applied for process modeling in a wide variety of domains such as computational, business, and real-time systems. A proper functionality of such systems can be achieved by verifying their designs and validating them according to their requirements.

In this chapter, we apply formal verification on UML activity diagram models using model-checking [45] due to the robust verification results provided. Model-checking is widely used in verification of software and hardware systems. It is an automated technique used to verify the functional requirements of behavioral models.

In this work, we were initially inspired by Eshuis et al. [21], but conceived a different algorithm that has some interesting implementation advantages from an object oriented perspective. As a result of our work, we developed an automated tool that retrieves the activity diagram from Artisan Real-time Studio. The diagram is converted to a configuration transition system that can be graphically visualized using an external graph drawing tool such as daVinci [11]. The latter can be used to provide a visual appraisal of the diagram complexity with respect to the number of nodes and edges. It can also be used as a quick feedback when applying corrective measures giving some insights about the resulting increasing or decreasing with respect to the diagram complexity. Furthermore, the NuSMV [15] model checker is invoked automatically. Finally, the tool displays the interpreted model-checking results providing the identified counterexamples if any.

In the following, we outline the informal syntax and semantics of UML activity diagrams. Also, we enumerate the system properties that we target in our model-checking procedure and give a brief overview of the used model checker and explain its temporal logic notation. Thereafter, we present the process for the verification and validation of the functional requirements captured using activity diagrams. Subsequently, we detail the algorithm that is used in order to achieve a model that is verifiable using model-checking techniques. Moreover, we present an example that covers concurrency, cross-synchronization, branching and merging points and give the analysis for the model-checking results. The chapter concludes by a general discussion of the effectiveness of the model-checking procedure.

4.2 UML 2.0 Activity Diagrams Syntax

UML activity diagrams [35] are used to depict the sequence of activities in a system. Activity diagrams are useful to show the workflow in a process from the start to the termination points. An activity diagram may contain many processing paths that consist of decision making and parallel processing. Moreover, the activity diagram shows the behavior of an entity from the perspective of activities or action states. Activity diagrams include elements that show the behavior of a system using control and dataflow models. In our work, we target the activity diagram artifacts that govern the control flow in the diagram. These artifacts are shown in Figure 4.1. An activity diagram consists of the following control flow elements:

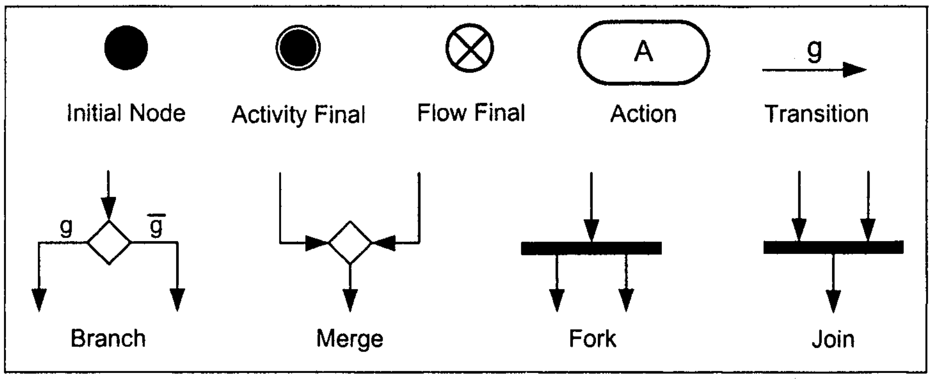


Figure 4.1: UML Activity Diagram Elements

- *Initial* node indicates the beginning of execution of a specific activity diagram.
- *Activity final* node terminates the execution in the whole activity diagram.
- *Flow final* node stops the execution of actions in the specified branch only.
- *Action* node comprises the process. Actions can be executed in chronological order or concurrently by the existence of forks. Furthermore, The action is where the execution is shown and it cannot be decomposed into smaller actions.

- *Branch* node is used to direct the control flow in the diagram into a specific execution path depending on the value of a guard.
- *Merge* node is used to merge the several paths selected by one or more branching nodes.
- *Fork* node is used to enable concurrent processing in parallel execution paths.
- *Join* node is used to synchronize different concurrent execution paths into one execution path.
- *Transition* is used to enable the flow of control in the diagram from an activity node to another. A transition can have a guard value that controls its control flow transfer.

An activity diagram may also contain object nodes for capturing object flows in the system. In this work, we covered the control flow aspects of the activity diagram.

4.3 System Properties

Verification and validation contribute to the design assessment by detecting the unsatisfied properties. Hence, system developers will know if the design is flawed and apply corrective measures. The following properties fall in the scope of our V&V approach for the model-checking of UML activity diagrams:

- *Latency*: It is the measure of the temporal delay between the request for the execution of an operation and the reply to this request. Detecting latency contributes to V&V by analyzing the efficiency of the system.

- *Liveness*: It asserts that under certain conditions, a given event will occur. It is known as "something good will always happen". Liveness analysis consists of checking whether some important or crucial events may or may not eventually happen in the system.

- *Safety*: It means that nothing bad can occur with respect to the design of the system. In other words, it is a judgment of the acceptability of risk, which implies that no harm will occur under the specified conditions.

- *Deadlock*: It describes a state wherein a process is waiting for some event that will never happen. It could be waiting for a resource to be available before continuing its execution while another process is holding indefinitely this resource. In this situation, the system would not progress.

- *Livelock*: It is a situation where two or more processes continually change their states in response to changes in the other process (or processes) without performing useful services. It is different from deadlock since the processes are progressing.

- *Precedence*: It specifies the order between events in the system with respect to time. Namely, events must not occur unless a specific event or a sequence of events were finished. If the ordering of events is not respected then V&V will help the developers review the ordering between events in their design.

- *Reachability*: It consists of checking whether a particular state is reachable in a design, starting from an entry point of the system. Unreachable states negatively impact the quality design since they denote dead entities in the system.

The aforementioned properties are supported by our selected model-checker. In the following, we give a glimpse of the NuSMV [15] model checker and the motivations to select it.

4.4 Model-checking of Activity Diagrams

This section presents the model-checking procedure when applied to UML 2.0 activity diagrams. Herein, the proposed model-checker is introduced and is followed by a brief presentation of the temporal logic that it is using. Subsequently, the notion of configuration transition system is introduced along with an algorithm that converts a given activity diagram into its corresponding configuration transition system. The latter along with various properties expressed in temporal logic can represent an input to the model-checker.

4.4.1 NuSMV Model Checker

The NuSMV [15] is a new symbolic model-checker developed as a joint project between Carnegie Mellon University[1] (CMU) and Istituto per la Ricerca Scientifica e Tecnolgica (IRST)[2]. NuSMV is designed to be a well structured, open, flexible and documented platform for model-checking. NuSMV is an enhanced version of the original SMV [28] model checker. NuSMV provides the desired functionalities for checking reachability, deadlocks, CTL [17] (Computation Tree Logic) fairness constraints, invariants, and computation of quantitative characteristics. Additionally, it supports the functionality for the generation and inspection of counterexamples. Moreover, LTL (Linear Temporal Logic) is also supported via reduction to CTL model checking.

[1]Carnegie Mellon University, http://www.cmu.edu/

[2]Istituto per la Ricerca Scientifica e Tecnolgica, http://irst.itc.it/

In this work, we use the NuSMV feature that targets finite state systems that are checked against properties specified in the CTL temporal logic. The input language of the NuSMV is used to describe the transition relations of a finite Kripke structure. Given that NuSMV is intended to describe finite state machines, the data types in the language are restricted to finite ones, i.e. boolean, scalar and fixed arrays of basic data types.

4.4.2 Computational Tree Logic

CTL [17] is a temporal logic that can be used to express properties of a system in the context of formal verification or model-checking. It uses atomic propositions and boolean connectives as its building blocks to make statements about the states of a system. It also uses temporal operators and path quantifiers.

CTL notation uses logical operators such as $\neg$ (negation), $\vee$ (or), $\wedge$(*and*), $\rightarrow$ (implies), $\leftrightarrow$ (equivalence) and uses the boolean constants *true* and *false*. The temporal operators that are used in the CTL notation are:

- *Next* (N ϕ) or known as (X ϕ) : ϕ has to hold at the next state.
- *Globally* (G ϕ): ϕ has to hold on the entire subsequent path.
- *Finally* (F ϕ): ϕ eventually has to hold (somewhere on the subsequent path).
- *Until* (ϕ U ψ): ϕ has to hold until at some position where ψ holds. This implies that ψ will be verified in the future.

- *Weak until* (ϕ W ψ): ϕ has to hold until ψ holds. The difference with U is that there is no guarantee that ψ will ever be verified.

Along with the aforementioned temporal operators, we have the universal and respectively the existential path quantifiers:

- *All* (A ϕ): ϕ has to hold on all paths starting from the current state.
- *Exists* (E ϕ): there exists at least one path starting from the current state where ϕ holds.

Furthermore, the temporal operators are always combined with a path quantifier as this is a requirement of the CTL model checking procedure.

4.4.3 Configuration Transition System

Any system that exhibits a dynamic of some kind can be abstracted to one that evolves within a discrete state space. Such a system is able to evolve through its state space assuming different configurations. A configuration c is a particular snapshot in the evolution of a set of elements of a system at a particular point in time and from a particular view.

Definition 1. (Configuration) *A configuration c is a specific binding of a set of values to the set of variables in the dynamic domain of a particular diagram.*

Informally, a configuration is specific to a particular type of UML behavioral diagram. For instance, a configuration for an activity diagram is the set of running actions at a given moment. In this case, we have a binding of boolean values to the set of actions in the diagram. That is, running actions are assigned *true* values, while inactive ones are assigned *false* values. The dynamic of the activity diagram can be captured by specifying all the possible configurations of the system and the transitions among them. Therefore, a UML activity diagram can be characterized by a set of configurations and a transition relation. Thus, we define a kind of a transition system that we called a Configuration Transition System (CTS).

Definition 2. (Configuration Transition System) *A CTS is a tuple* $(C,\Lambda,\rightarrow)$*, where* C *is a set of configurations taken from the same view,* Λ *is a set of labels, and* $\rightarrow \subseteq C \times \Lambda \times C$ *is a ternary relation, called a transition relation. If* $c_1, c_2 \in C$ *and* $l \in \Lambda$*, the common representation of the transition relation is:* $c_1 \xrightarrow{l} c_2$.

Since the dynamics of activity diagrams can be captured by the corresponding CTS, we can consider it as the diagram semantics model. Thus, the CTS can be used to systematically generate the model-checker input.

4.4.4 Generation of Configuration Transition System

Given an instance of an activity diagram, we can find the corresponding configuration system provided that the elements of the diagram are understood and there exists (and is defined) a step relation that enables one to compute the next configuration(s) of a diagram from the current one. Furthermore, in order to achieve tractability, the configuration space should be bounded.

In order to efficiently generate the CTS for a given activity diagram, each of the configurations enclosed within a configuration system represents a set of states that are active simultaneously, a set of guard values and a set of joint patterns. The latter is understood as a list of action nodes that finished their execution and are waiting to join at a synchronization point which represents the *join* node in activity diagrams.

In the sequel, we explain Algorithm 1 that is used to generate the CTS for activity diagrams. In order to generate the CTS, we need the following data structures:

- *confList*: the list of found configurations.
- *crtConf*: the list of active actions in the diagram.
- *transList*: the list of found transitions.
- *container*: contains all the activity diagram nodes, the current guard values and the current join pattern list.
- *actConfList*: the list of newly identified configurations.

The CTS is obtained by a Breadth-first search iterative procedure that explores new configurations from a current configuration *crtConf*. Each configuration is understood as a list of three elements which are in order, the list of active states in this configuration (first element), the list of guard values (second element) and the list of join patterns (third element) respectively. In a configuration transition system, each transition is a list of two elements which are in order the source configuration and the target configuration.

Initially, *confList* and *transList* are empty. The container is initialized with the diagram state list, the initial guard value list, and an empty join pattern list. Whenever the values of a guard are not fixed they will be assigned the generic *"any"* value, meaning that the guard can be either *true* or *false*. *actConfList* is initialized with the result of the *getConf* procedure, presented in Algorithm 2, applied to the container. The *getConf* procedure returns the configuration corresponding to the

Algorithm 1 Generation of Configuration Transition System

```
ActConfList confList = {}
ActTransList transList = {}
ActConfiguration container = {DiagramStateList,guardValueList,{}}
ActConfList actConfList = getConf(container)

while actConfList is not empty do
  ActConfiguration crtConf = pop(actConfList)
  ActStateList crtStateList = get(crtConf,0)
  GuardList crtGList = get(crtConf,1)
  ActJoinPatList crtJoinPatList = get(crtConf,2)

  if crtGList contains "any" then
    splitIndex = getPosition(crtGList, "any")
    crtGList[splitIndex] = true
    actConfList = actConfList ∪ { crtStateList, crtGList, crtJoinPatList }
    crtGList[splitIndex] = false
    actConfList = actConfList ∪ { crtStateList, crtGList, crtJoinPatList }
    continue
  end if

  if confList not contains crtConf then
    confList = confList ∪ crtConf
  end if

  for each state s in crtStateList do
    setConf(container, crtConf)
    execute(s)
    nextConf = getConf(container)

    if nextConf not equals crtConf then
      actConfList = actConfList ∪ nextConf
      crtTrans = {crtConf, nextConf}

      if transList not contains crtTrans then
        transList = transList ∪ {crtTrans}
      end if

    end if

  end for

end while
```

Algorithm 2 getConf(container)

```
ActStateList stateConf = {}
for each state s in container do
  if s is active then
    stateConf = stateConf ∪ {s}
  end if
end for
return { stateConf, get(container,1), get(container,2) }
```

Algorithm 3 setConf(container, crtConf)

```
crtConfStateList = get(crtConf,0)
containerStateList = get(container,0)
for each state s in containerStateList do
  if crtConfStateList contains s then
    setActivate(s)
  else
    setInactivate(s)
  end if
end for
container = {containerStateList , get(crtConf,1), get(crtConf,2)}
```

current state of the container. Subsequently, *actConfList* is iterated in a while loop until it becomes empty. In every iteration, the first element of *actConfList* is popped and becomes the current configuration *crtConf*, containing a current state list *crtStateList*, a current guard list, *crtGList* and a current join pattern list, *crtJoinPatList*. If *crtGList* contains an *"any"* value, then two new configurations are created and added to *actConfList*, for each of the possible guard values, and thereafter, the next iteration immediately begins. Otherwise, *confList* is updated with *crtConf* if it does not already contain it, followed by a *"for"* loop that iterates the states of *crtStateList*. In every *"for"* loop, the iterated state is executed after the container state is set to the one of *crtConf* by calling the *setConf* procedure, presented in Algorithm 3. Thereafter, *nextConf* is assigned the result of *getConf* procedure applied to the container. If *nextConf* is not equal to *crtConf*, then *nextConf* is added to *actConfList* and likewise, a transition from *crtConf* to *nextConf* is added to *transList*.

In contrast to the approach of Eshuis et al. [21] the foregoing algorithm does not require the presence of wait states in order to synchronize activity nodes. Furthermore, it does not require the conversion of the activity diagram to the intermediary form of activity hypergraph.

Additionally, the approach proposed here is compliant with the UML 2.0 activity diagram execution semantics even though it does not make explicit use of tokens. However, in an equivalent manner, the action states can transfer control. Moreover, they can synchronize if required, by recording a join pattern list in every configuration of the configuration transition system.

4.5 Architecture

The UML 2.0 activity diagram assessment tool is comprised from several components arranged in three layers. At the front-end layer, we have the modeling tool, the GUI application and the graph viewing component for displaying the configuration transition system (CTS). At the middleware layer, we have the ActiveX component that communicates with the modeling tool datastore in order to retrieve the diagram information, the NuSMV model checker and the Java engine responsible for the CTS and the NuSMV code generation. Finally, the Back-end layer consists of the modeling tool datastore and the file system. Figure 4.2 depicts the three layers and the interactions among them.

The tool has a plug-in in the modeling tool to enable the user to apply the assessment on the current activity diagram. The user interface layer calls the *ActiveX* to retrieve the activity diagram from the *datastore*. After the model is fetched, the *java engine* is executed to generate the *Configuration Transition System* (CTS) and the *NuSMV code* for the model-checker. Thereafter, the NuSMV model checker is run by clicking the *check properties* button to perform the assessment operation. After the operation is complete, the NuSMV model checker results are placed in the file system. In addition to the manually specified properties, the deadlock and reachability properties are checked automatically by the model-checker without any user intervention. Finally, the verification tool reads the model-checking results and interprets them in order to be understood by the user.

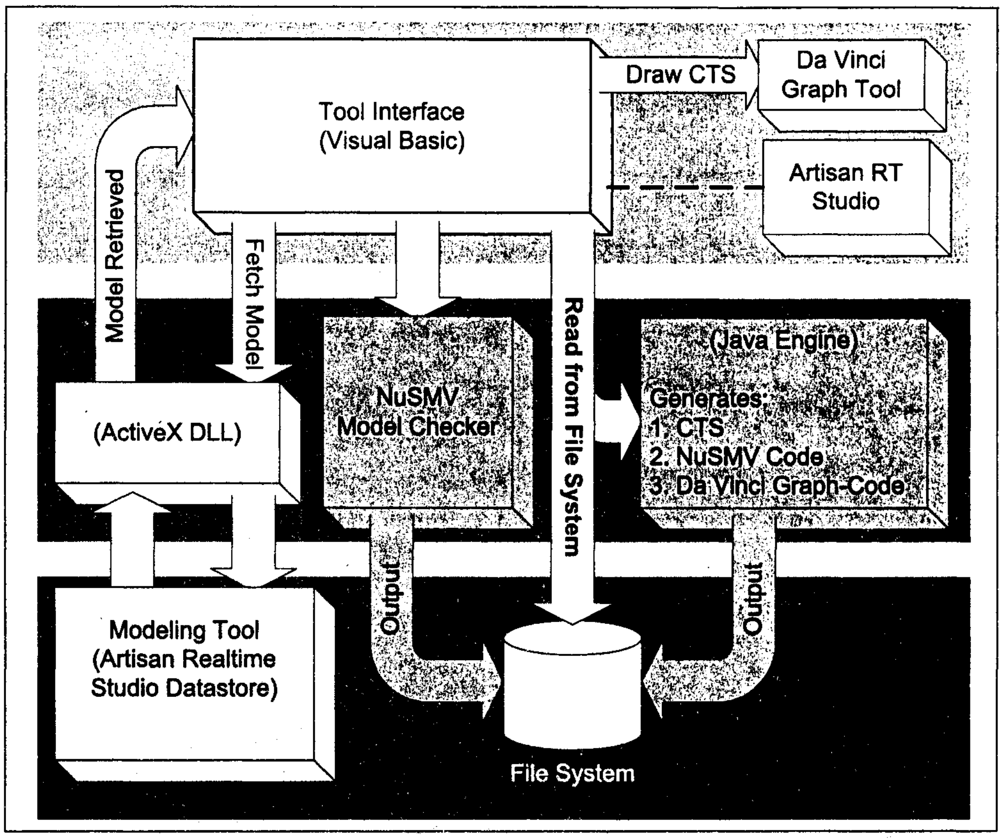
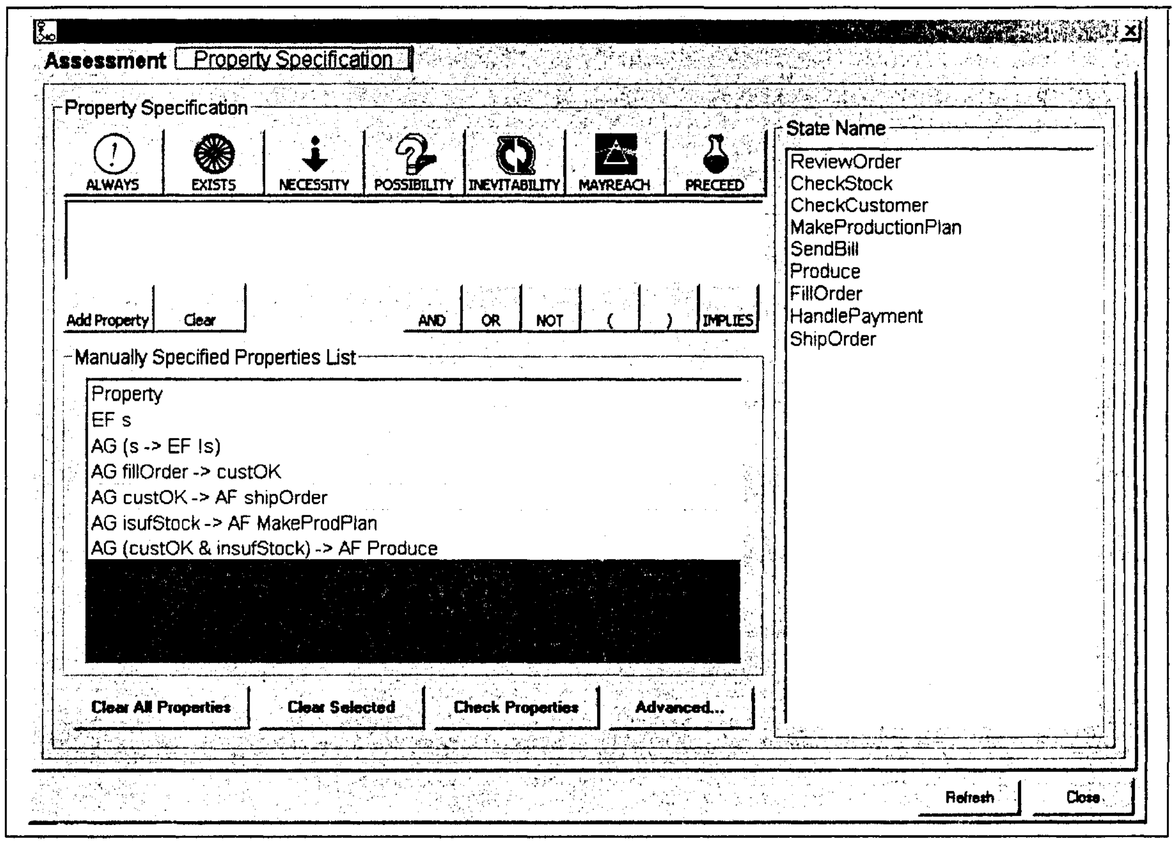


Figure 4.2: Architecture of the Activity Diagram Verification Tool

4.5.1 User Interface

Figure 4.3 presents the user interface panel that is used to specify properties and run the model-checker on the current activity diagram. This interface is designed in a flexible manner to enable a user that is not familiar with model-checking commands to specify and customize some properties to be verified against the system requirements.

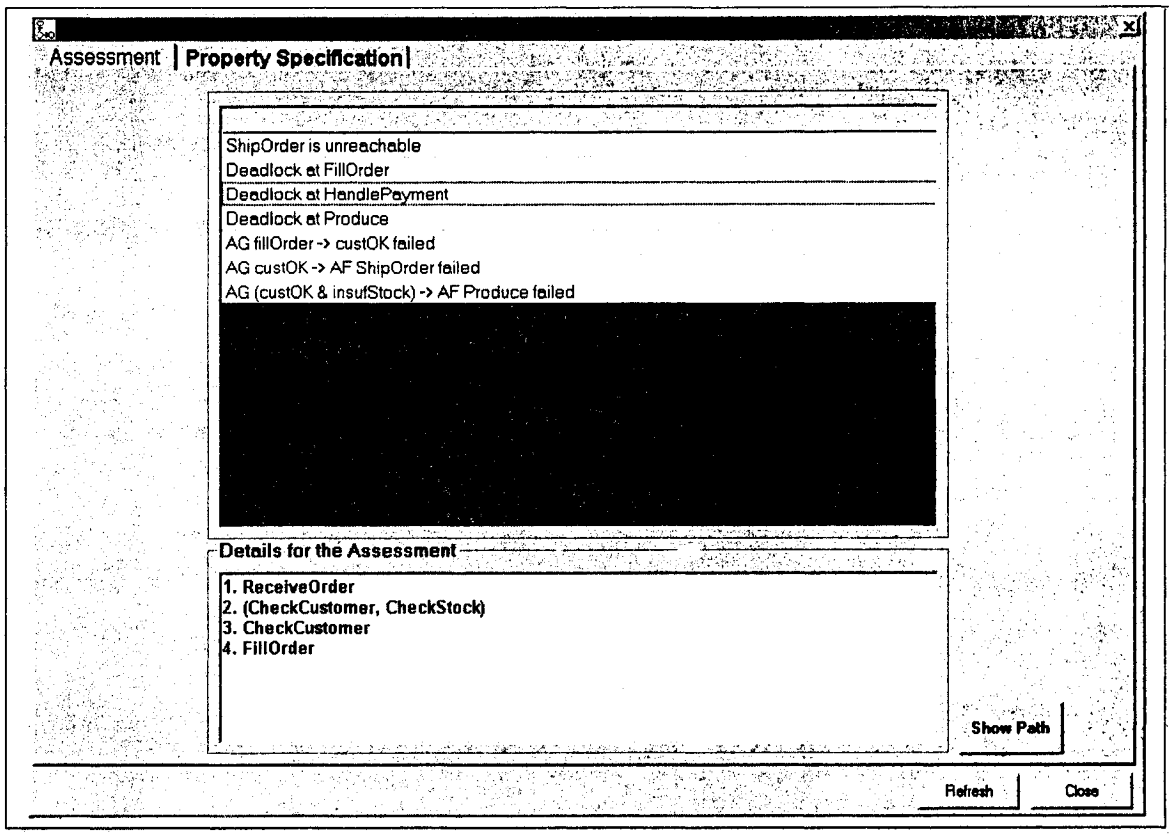


Property
EF s
AG (s -> EF Is)
AG fillOrder -> custOK
AG custOK -> AF shipOrder
AG isufStock -> AF MakeProdPlan
AG (custOK & insufStock) -> AF Produce

ReviewOrder
CheckStock
CheckCustomer
MakeProductionPlan
SendBill
Produce
FillOrder
HandlePayment
ShipOrder

Figure 4.3: Snapshot from the GUI Tool (Specification of Properties)

Figure 4.4 depicts the second user interface panel that shows the assessment results for the model-checking process. The assessment results are listed along the counterexample for each failed property in the *details for the assessment* box. The tool also provides a visual assessment checking using a graph drawing tool namely *Da Vinci.*



ShipOrder is unreachable
Deadlock at FillOrder
Deadlock at HandlePayment
Deadlock at Produce
AG fillOrder -> custOK failed
AG custOK -> AF ShipOrder failed
AG (custOK & insufStock) -> AF Produce failed

1. ReceiveOrder
2. (CheckCustomer, CheckStock)
3. CheckCustomer
4. FillOrder

Figure 4.4: Snapshot from the GUI Tool (Assessment)

This tool covers most of the logical and temporal operators found in CTL. The user should understand the meaning of these operators in order to be able to specify properties. These specified properties represent the system's requirements that should be verified in the design.

The user has the option to view the the counterexample for the selected failed property by click the *show path* button. Figure 4.5 shows an example of a failed property in Figure 4.4. The visual verification enables the user to identify the problem in the CTS which enriches his/her understanding of the problem. The diagram in Figure 4.5 represents a CTS with a deadlock that is represented in a *deadlock* which helps the user to locate the deadlock occurrence in the transition system. Moreover, the counterexample has a different color in order to track its path easily in the generated CTS.

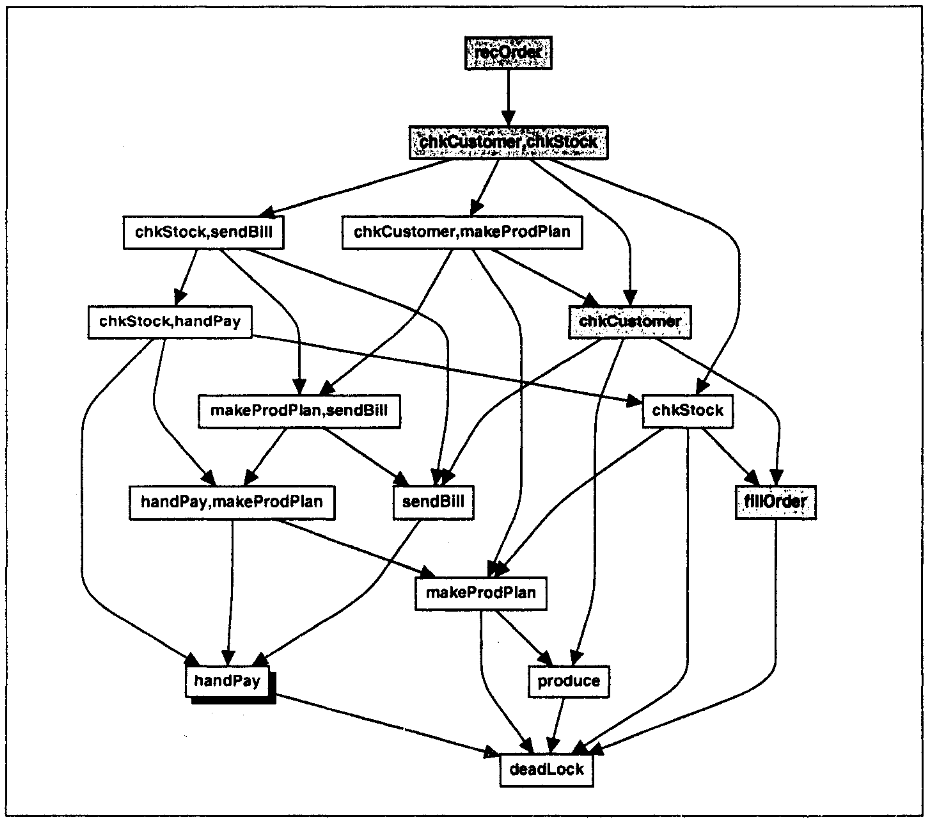


Figure 4.5: Snapshot for a Counterexample from Da Vinci Tool

4.5.2 Java Engine Design

Figure 4.6 shows the class diagram of the Java Engine presented in Figure 4.2. The main class is *ActState* which is used as the building block when representing the activity diagram. It has a number of subtypes corresponding to the various pseudo states such as branch, merge, fork, join and flow end. Each configuration is composed of a list of *ActState* objects, a stack of guard values that are read from the singleton instance of the *ActGStack* class, and an *ActJoinPatList* object which in turn is composed of activity join pattern objects, ActJoinPat. The singleton instance of the Act-DeadLock class is used to replace any *ActJoinPat* object in an *ActJoinPatList* that is never matched. The activity configuration system is contained in an *ActConfSys* object. The latter is composed of a list of activity configuration objects, *ActConfiguration*, and an activity transition list object, *Act-TransList*. Algorithm 1 makes use of the functionality presented within the implementation of the class methods.

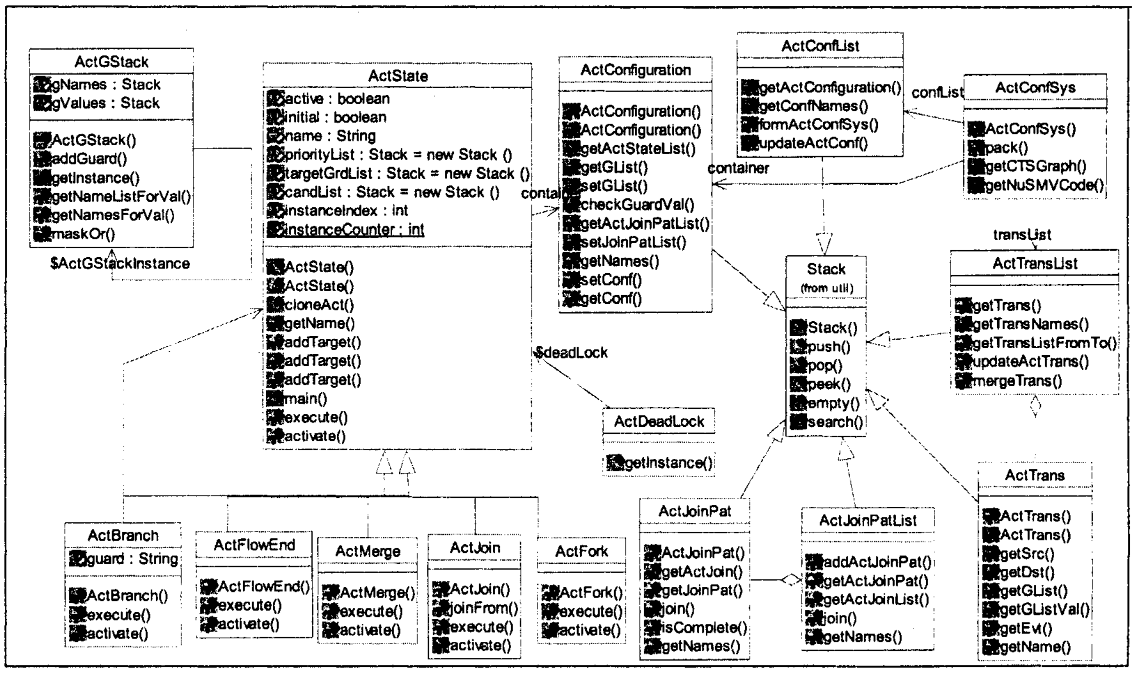


Figure 4.6: Class Diagram of the Java Engine Architecture

4.6 Case Study and Analysis

This section presents an activity diagram example that depicts a business workflow process. The activity diagram is depicted in Figure 4.7 and basically represents a purchase request order processing.

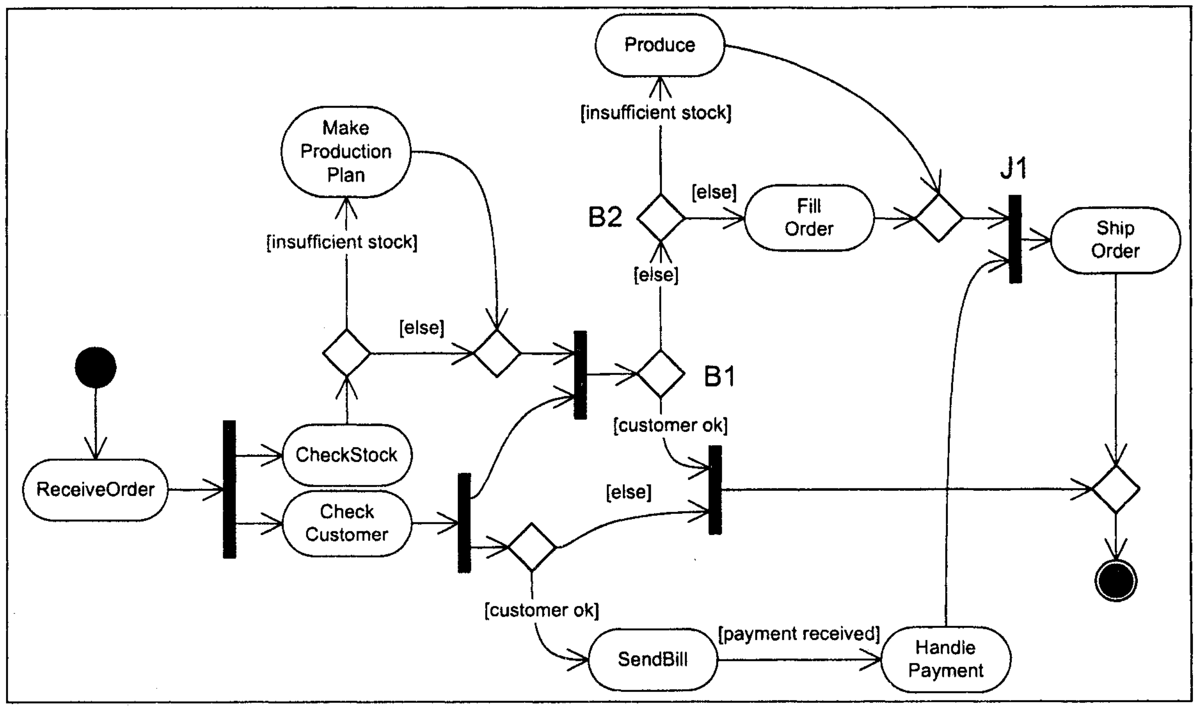


Figure 4.7: Activity Diagram Example

In order to assess the diagram, it is converted to its corresponding configuration transition system using Algorithm 1 presented is Section 4.4.4. The configuration transition system is then input to the NuSMV model checker along with the properties to be verified expressed as CTL formulas. The diagram intentionally contains a subtle flaw that will be detected in the assessment procedure. Figure 4.8 depicts the generated transition system for the flawed activity diagram presented in Figure 4.7.

We are interested in two kinds of properties. On one hand, we have automatic specifications that check for reachability and absence of the deadlock in the diagram. Thus, for every state s of the diagram we automatically generate the following CTL properties:

- s should be reachable. The reachability property is expressed in CTL as follows:

  EF s

- s should be deadlock free. The deadlock free property is expressed in CTL as follows:

  AG (s $\rightarrow$ EF !s)

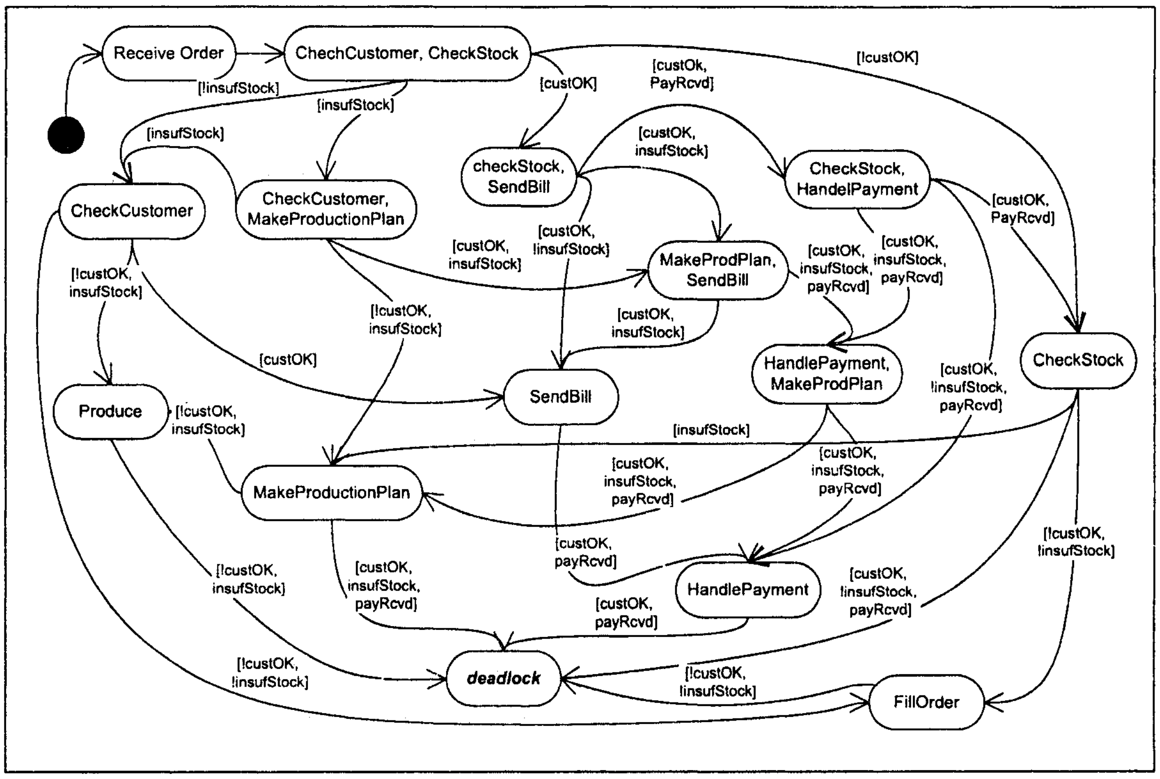


Figure 4.8: Corresponding CTS for the Flawed Activity Diagram

On the other hand, we have the manually specified properties that are intended to validate the behavior of the diagram. While the deadlock and reachability properties are specified for every state of the diagram, the manual specifications may involve only some of the states of the diagram as they target the fulfillment of the design requirements.

Thus, for the presented activity diagram we give an example of some interesting safety and liveness properties. A safety property basically states that nothing bad can happen, whereas the liveness property states that something good will eventually happen.

- The requirements state that it is always the case that the customer is trusted when *fillOrder* action is reached. Below is the CTL notation for this safety property:

  ```
  AG fillOrder → custOK
  ```

- Whenever the customer is trusted there should be a *ship order* activity state reached. Below is the CTL notation for this liveness property:

  ```
  AG custOK → AF shipOrder
  ```

- Whenever there is insufficient stock a production plan activity should be eventually reached. Below is the CTL notation for this liveness property:

  ```
  AG insufStock → AF MakeProdPlan
  ```

- If the customer is trusted and there is insufficient stock the Produce activity should eventually be reached. Below is the CTL notation for this liveness property:

  ```
  AG (custOK & insufStock) → AF Produce
  ```

After generating the CTS, we compile the corresponding NuSMV code and add the aforementioned automatic and manual property specifications in the form of CTL formulas. For the flawed activity diagram the following properties failed:

- The *ShipOrder* node was unreachable.
- A deadlock was identified in the diagram because the node J1 was unable to proceed as it was waiting indefinitely for an incoming joining transition. Below is a counterexample generated by the model-checker:

  `ReceiveOrder` $\rightarrow$ `(CheckCustomer,CheckStock)` $\rightarrow$

  `CheckCustomer` $\rightarrow$ `FillOrder`

- The CTL property `AG fillOrder` $\rightarrow$ `custOK` failed due to the wrong value of the guard in the branching point B1.
- The CTL property `AG custOK` $\rightarrow$ `AF shipOrder` failed as *ShipOrder* was unreachable in the diagram.
- The CTL property `AG (custOK & insufStock)` $\rightarrow$ `AF Produce` failed due to the wrong value of the guard that is tested in the branching point B1.

The CTS diagram in Figure 4.8 shows a deadlock state that is reached on all the execution paths of the CTS. This is caused by the fact that the value that is checked against the guard of the transition going from the branching node B1 to the branching node B2 has the wrong value. After fixing this issue and rerunning the model-checker on the corrected model all the properties both automatic and manually specified passed. The CTS graph for the corrected activity diagram is shown in Figure 4.9. As can be seen from the figure, the *deadlock* state that was in the flawed version has disappeared.

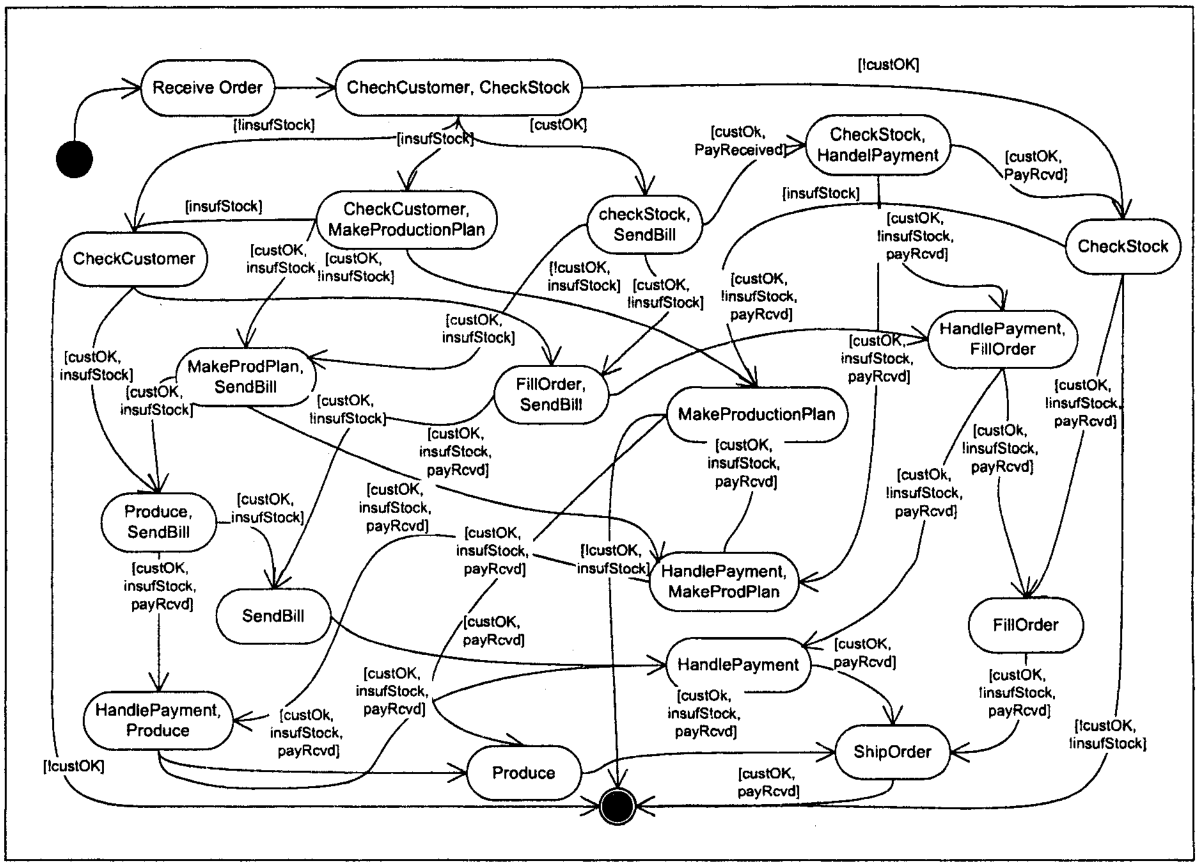


Figure 4.9: Corresponding CTS for the Corrected Diagram

4.7 Model-Checking Feasibility

In order to assess the model-checking procedure feasibility, several experiments were performed during the ongoing research efforts that were concretized with the publication of the results in

[2–4]. We briefly present some results obtained from different experiments with respect to the model-checking procedure. The time required to obtain the model-checking results for different activity diagrams was related to the complexity of their CTS as expected. The experiments were conducted on a 2.6 GHz P4 workstation with 1G of RAM. In the cases where the number of states of the CTS was around one hundred or less, the model-checker computed the results pretty fast ranging from less than a second to less than a minute. However, when the number of states was very large, the results were computed in intervals peeking to several hours while the memory usage was measured in hundreds of megabytes.

One of the situations that increased the computation time was represented by the transitions that formed loops in the CTS. This fact resulted in a larger number of execution traces especially when combined with extensive forking. In order to be more effective, the experiments were stopped at the point were excessive memory space was required. It must be understood however that the usage of forking was literally abused in order to push the model-checker in significantly consuming computational resources in terms of processor cycles and memory space. This was only meant to evaluate the limits that might be encountered for certain larger size systems where the number of concurrent activities might be elevated. Nevertheless, even industry sized models would hardly reach a critical limit that would preclude the model-checker from finishing in a reasonable amount of time.

Even though the model-checking technique might be memory hungry and time consuming for models with a very large number of states and intricate transitions, it has the advantage of exhaustively verifying the model against the desired properties. Moreover, the designers are usually interested in a moderate number of execution traces that reflect the design requirements. Hence, we can safely say that a design exhibiting a wild complexity might be reconsidered as long as the number of unimportant execution traces outweighs the number of the important ones. It follows that even a large model that entails a moderate number of execution traces can be subject to model-checking with a significant degree of success.

In contrast, for the same model, traditional simulation might miss some subtle corner cases that were not covered by the selected test vectors. Though it is conceivable that one may find all the test vectors required to thoroughly complete the simulation of a particular model, the effort required for achieving such an objective is typically unfeasible. Moreover, it is often the case that even simulators require tremendous resources in terms of CPU cycles and memory space. It is not uncommon, especially in the industry, to have impressive high end hardware that is used to run very long regressions. It is obvious that formal methods like model-checking might also benefit from such high end hardware. There are even model-checkers that are able to benefit from clusters and distributed computing [57].

4.8 Summary

In this chapter, a new algorithm for the formal verification and validation of UML 2.0 activity diagrams was presented. Moreover, an activity diagram example was assessed by the generation of its corresponding CTS. The CTS is then translated to the SMV code which is input to the NuSMV model checker. Finally, a discussion of the feasibility of model-checking was presented along with supporting arguments inspired from our conducted experiments.

Chapter 5

Software Engineering Metrics in Systems Engineering

5.1 Introduction

The need for a reliable and a high performing software has led to the emergence of software engineering. Since the birth of software engineering in 1968, new approaches and techniques were developed to govern the quality of software systems. Software metrics are used to assess the quality of software systems in terms of system attributes such as complexity, understandability, maintainability, stability and others. Different software metrics were developed to measure the quality of structural and objected-oriented programming techniques. Some of the metrics for the structural programming are the Lines Of Code (LOC) and Cyclomatic Complexity (CC) [43] metrics. When the object-oriented paradigm emerged, many new metrics evolved to assess the quality of software system design and to overcome the limitations of the legacy code metrics.

UML [35] has been standardized as a modeling language for object-oriented systems. In this chapter, a set of object-oriented metrics were collected to measure the quality of UML class and package diagrams. In the following sections, we argument the relativeness of software engineering metrics to systems engineering and we present the quality attributes that are measured using the targeted metrics. Thereafter, a set of fifteen metrics for package and class diagrams are detailed. Finally, we present a snapshot of the developed tool and an analysis of a class and pacakge diagram case study.

5.2 Relevance to Systems Engineering

In the systems engineering arena, there are a number of initiatives [7, 62] that employ the object oriented design paradigm in the design of systems. These initiatives acknowledge and take advantage of the benefits of the object oriented approach. Thus, the software engineering field experience can be successfully imported in systems engineering design. Moreover, modern system design currently employs modularity which invites the design engineers to think of the similarity with software objects and components. Though there are many similarities in the structural and architectural perspectives, there are also peculiarities that prohibit the use of the full range of features present in software object oriented design. For example, commonly used techniques such as object creation and destruction, thread spawning and other software specific constructs can hardly be accommodated in the traditional systems design.

For this reason, the present draft [52] of the SysML specification confines the available features to a restricted set that can be readily applied in systems engineering design.

Many software engineering techniques were developed to assess the object oriented designs. In this context, the more a system design is object oriented, similar or even identical metrics can be applied in order to assess various system quality attributes.

Modern modeling languages for systems engineering such as UML 2.0 and SysML are used to capture the system requirements and to specify its components. Moreover, they can be used to model the behavior of the system and its components. Since these modeling languages are compatible with software systems, some existing software engineering techniques can be easily inherited or adapted if necessary.

Furthermore, any system exhibits a degree of complexity with respect to the relationships among its components. Thus it is important to evaluate the coupling among the components and the cohesiveness degree for each component. Also, many complex systems include both hardware and human resources. Therefore, the complexity of workflow within such a system requires rigorous means for analysis and assessment.

The research presented hereby aims to show the usefulness and relativeness of software engineering techniques and specifically software metrics to assess systems engineering designs.

5.3 Quality Attributes

There are many quality attributes that are captured using the set of object oriented metrics. In the sequel, we briefly present those that we target in this work:

- *Stability:* indicates the risk level of the occurrence of unexpected effects, occasioned by modifications on the software.

- *Understandability:* measures the degree to which the system stakeholders are able to comprehend the system specifications.

- *Maintainability:* measures the easiness and rapidity with which a system design and/or implementation can be changed for perfective, adaptive, corrective, and/or preventive reasons.

- *Reusability:* measures the easiness and rapidity with which a part (or more) of a system design and/or implementation can be reused.

- *Coupling:* measures how strongly system parts depend on each other. Generally, a loose coupling is sought in a high-quality design. Moreover, there is a strong correlation between coupling and other system quality attributes such as complexity, maintainability and reusability.

- *Cohesion:* refers to the degree to which system components are functionally related (internal "glue"). Generally, a strong cohesion is sought in a high-quality system design.

- *Complexity:* designates the quality of being intricate and compounded. It measures the degree to which a system design is difficult to be understood and/or to be implemented.

The aforementioned quality attributes are the cornerstone in building quality software systems. In the next section, we present the set of software engineering metrics that were used in our work to assess system quality attributes mentioned in this section.

5.4 Metrics Suite

This section discusses the set of fifteen metrics for class and package diagrams. Subsequently, we explain each metric separately and its formulas and the corresponding nominal range.

5.4.1 Abstractness

The Abstractness [42] metric measures the package abstraction rate. A package abstraction level depends on its stability level. Calculations are performed on classes defined directly in the package and those defined in sub-packages. In UML models, this metric is calculated on all the model classes. The Abstraction metric provides a percentage between 0% and 100%, where the package contains at least one class and at least one operation in an abstract class. The following formula is used to measure the abstractness of the package diagram.

$$Abstraction = \frac{N_{ma}}{N_{mca}} \times \frac{N_{ca}}{N_c} \times 100 \qquad (1)$$

where:

- N_{ma} is the number of abstract methods in all the package's classes.
- N_{mca} is the number of methods (abstract or not) in the package's abstract classes.
- N_{ca} is the number of abstract classes.
- N_c is the number of classes (abstract or not) of the package.

The Abstractness metric depends on how a package is subject to modification during the life cycle of the application. A package should be more abstract in order to be extensible, which means a more stable package. Abstract packages which are extensible provide greater model flexibility.

Nominal values for this metric can not be measured since abstractness depends on the purpose of the package.

5.4.2 Instability

The Instability [42] metric measures the level of instability in a package. A package is unstable if it depends more on other packages than they depend on it. The instability of a package is the ratio of its afferent coupling to the sum of its efferent and afferent coupling and is measured using the following formula.

$$I = \frac{AC}{EC + AC} \tag{2}$$

Where:

- AfferentCoupling (AC) is the number of links (associations, dependencies and generalizations) towards classes defined in other packages.
- EfferentCoupling (EC) is the number of links (associations, dependencies and generalizations) coming from classes defined in other packages.

A package is more likely to be subject to change if the other packages, that it depends on, change. The instability metric does not have a nominal value since some packages must be kept unstable to enable them for extensibility.

5.4.3 Distance from the Main Sequence

The *DMS* [42] metric measures the appropriate balance between the abstraction and the instability rates of the package. Packages should be quite general in order to be consistent with these two orthogonal criteria. Packages should be unstable to be subject to modifications. However, packages should have some level of abstraction. Therefore, a balance should be achieved between the package's abstraction and instability and can be measured by the following formula.

$$DMS = |Abstraction + Instability - 100| \tag{3}$$

A *DMS* of 100% has an optimal balance between abstraction and instability. Practically, a value greater than 50% is considered to be within the nominal range of *DMS*.

5.4.4 Class Responsibility

The *CR* [24] ratio is an indication of responsibility level assigned for each class in order to correctly execute an operation in response to a message. A method is considered to be responsible if it has pre-conditions and/or post-conditions. A class method should be responsible to check whether a message is appropriate before taking any action. On the other hand, a class method should take responsibility to ensure the success of the method.

CR is a ratio of the number of methods which implement pre-condition and/or post-condition contracts to the total number of methods. *CR* is calculated using the following formula.

$$CR = \frac{PCC + POC}{2 \times NOM} \times 100 \tag{4}$$

Where:

- *PCC* is the total number of methods which implement pre-condition contracts.
- *POC* is the total number of methods which implement post-condition contracts.
- *NOM* is the total number of methods.

The *CR* nominal range is between 20% and 75%. A value below 20% indicates irresponsible class methods. Irresponsible methods indicate that the class will passively react to the sent and received messages. Responsible methods are desired as they diminish the number of runtime exceptions in a system. A CR value above 75% is preferable but seldom achieved.

5.4.5 Class Category Relational Cohesion

The *CCRC* [24] metric measures how cohesive are the classes in the diagram. The construction of classes in the diagram must be justified by the links that exist between its classes. In the class diagram, a scarce level of relations among the classes indicates a lack of cohesiveness. Relational cohesion is the number of relationships among classes in the class diagram divided by the total number of classes in the diagram. *CCRC* is calculated using the following formula.

$$CCRC = \frac{\sum_{i=1}^{N_c} NA_i + \sum_{i=1}^{N_c} NG_i}{N_c} \times 100 \qquad (5)$$

Where:

- NA is the number of association relationships for a class.
- NG is the number of generalization relationships for a class.
- N_c is the number of classes in the diagram.

A class is considered to be cohesively related to other classes in the class category when this class collaborates with other classes to achieve its responsibilities. A *CCRC* value less than 1 indicates that some classes have no relationships with any other class in the model. A *CCRC* nominal range is between 150% and 350%. A value greater than 350% is not preferable due to complexity considerations.

5.4.6 Depth of Inheritance Tree

Inheritance is an important concept in object oriented models, however it should be carefully used to achieve the goals of a good software system design. Classes that are located deep in the inheritance tree are more complex and difficult to develop, test and maintain. To achieve a good system design, a trade-off should be considered in creating the class hierarchy. Thus, it was empirically found that a *DIT* value between 1 and 4 fulfills this goal while a value greater than 4 would increase the complexity of the model.

Algorithm 1 Measuring the Depth Of Inheritance (DIT) of a Class

```
global integer DITMax = 0
for each class c in CD do
   call TraverseTree(c)
end for
function TraverseTree(class c)
{
static integer DIT = 0
for each generalization relationship g from class c do
   get superclasses of c
   for each superclass s of class c do
      DIT = DIT + 1
      TraverseTree (s)
   end for
   if DIT GT DITMax then
      DITMax = DIT
   end if
   DIT = DIT - 1
end for
}
```

Algorithm 1 is used to measure the depth of inheritance. The recursive method *TraversTree* checks for the depth of inheritance for each class. The algorithm iterates all the classes in the diagram and records the maximum inheritance depth in *DITMax*.

5.4.7 Number of Children

The *NOC* [14] metric measures the average number of children for the classes in the class model. *NOC* is important due to the following factors:

- A large number of children indicates that a larger degree of reuse is achieved.
- Too many children may indicate a misuse of subclassing which will increase the complexity.

NOC is calculated by summing the number of children for each class in the model. Then, this number is divided by the total number of classes except for the child classes at the lowest level in the model. *NOC* is calculated using the following formula.

$$NOC = \frac{\sum_{i=1}^{N_c} NCC_i}{N_c - LLC} \tag{6}$$

Where:

- *NCC* is the sum of children for a class.
- N_c is the total number of classes in the diagram.
- *LLC* is the number of classes in the lowest level of inheritance in the diagram.

An *NOC* value of 0 shows that this is a non object-oriented model. A nominal range for *NOC* is between 1 and 4. A value in this range indicates that the goals of reuse are compliant with the goals of managing complexity and promoting encapsulation. A number greater than 4 may indicate improper abstraction.

5.4.8 Coupling Between Object Classes

CBO [14] is a measure of the average degree of connectivity and interdependency between objects in a model. It is directly proportional to coupling and complexity, and inversely proportional to modularity. Therefore, it is desired to have a lower *CBO* value. This value is important due to the following reasons:

- Strong coupling inhibits the possibilities of reuse.
- Strong coupling makes a class difficult to understand, correct, or change without related changes to other classes in the model.
- Tight coupling increases the model complexity.

CBO is calculated using the following formula.

$$CBO = \frac{\sum_{i=1}^{N_c} AR_i + \sum_{i=1}^{N_c} DR_i}{N_c} \tag{7}$$

Where:

- *AR* is the total number of association relationships for each class in the diagram.
- *DR* is the total number of dependency relationships for each class in the diagram.
- N_c is the number of classes in the diagram.

A *CBO* value of 0 indicates that a class is not related to any other classes in the model and therefore should not be part of the system. The nominal range for *CBO* falls between 1 and 4 indicating that the class is loosely coupled. A *CBO* value above 4 may indicate that the class is tightly coupled to other classes in the model therefore complicating the testing and modification operations and limiting the possibilities of reuse.

5.4.9 Number Of Methods

NOM [38] metric is the average count of methods per class. The number of methods in a class should not be high but not at the expense of missing or incomplete functionality. This metric is useful in identifying classes with little or no functionality thus serving mainly as data types. Moreover, a subclass that does not implement methods has little or no potential for reuse.

This metric is measured by counting the total number of methods (defined and inherited from all parents) for all the classes in the model. Then, this number is divided by the total number of classes in the model. Thus, *NOM* is calculated using the following formula.

$$NOM = \frac{\sum_{i=1}^{N_c} NM_i + \sum_{i=1}^{N_c} NIM_i}{N_c} \tag{8}$$

Where:

- NM is the number of methods for a class.
- NIM is the total number of inherited methods by a class.
- N_c is the number of classes in the diagram.

The *NOM* nominal range lies between 3 and 7 and indicates that the class has a reasonable number of methods. An *NOM* value greater than 7 indicates the need for decomposing the class into smaller classes. Alternatively, a value greater than 7 may indicate that the class does not have a coherent purpose. A value less than 3 indicates that a class is merely a data construct rather than a true class.

5.4.10 Number Of Attributes

The *NOA* [40] metric measures the average number of attributes for a class in the model. This metric is useful in identifying the following important issues:

- A relatively large number of attributes in a class may indicate the presence of coincidental cohesion. Therefore, the class needs to be decomposed into smaller parts in order to manage the complexity of the model.

- A class with no attributes means that a thorough analysis must be done on the semantics of the class or may indicate that it is a utility class rather than a regular class.

NOA is the ratio of counting the total number of attributes (defined and inherited from all ancestors) for each class in the model to the total number of classes in the model. *NOA* is calculated using the following formula.

$$NOA = \frac{\sum_{i=1}^{N_c} NA_i + \sum_{i=1}^{N_c} NIA_i}{N_c} \tag{9}$$

Where:

- NA is the total number of attributes for a class in the diagram.
- NIA is the total number of inherited attributes for a class in the diagram.
- N_c is the number of classes in the diagram.

A nominal range for *NOA* falls between 2 and 5. A value within the nominal range indicates that a class has a reasonable number of attributes whereas a value greater than 5 may indicate that the class does not have a coherent purpose and requires further object-oriented decomposition. A value of 0 for a particular class may designate it as a utility class.

5.4.11 Number of Methods Added

The *NMA* [40] metric plays a significant role in the assessment of the class specialization. A class with too many added methods indicate an overspecialization when compared to the functionality of its ancestors. Consequently, inheritance would be rendered less effective due to the major differences between the subclass and its ancestors. *NMA* is the ratio of all the added methods in the diagram to the total number of classes in the diagram. *NMA* is computed using the following formula.

$$NMA = \frac{\sum_{i=1}^{N_c} AM_i}{N_c} \tag{10}$$

Where:

- MA is the total number of added methods for a class.
- N_c is the number of classes in the diagram.

This metric has a nominal range between 0 and 4. A value greater than 4 indicates that a class has major changes from its ancestors. A class with an *NMA* value above 4 inhibits the use of inheritance.

5.4.12 Number of Methods Overridden

NMO [40] metric plays a significant role in the assessment of the class specialization. A class with too many redefined methods implies that little or no functionality is reused which may indicate misuse of inheritance. *NMO* is the count of the number of redefined methods in the class and is calculated as follows.

$$NMO = \frac{\sum_{i=1}^{N_c} RM_i}{N_c} \tag{11}$$

Where:

- RM is the total number of redefined methods in a class.
- N_c is the number of classes in the diagram.

A class that inherits methods must use them with the minimum of modifications. A class with a high number of redefined methods yields to a loss in the meaning of inheritance. This metric has a nominal range between 0 and 5.

5.4.13 Number of Methods Inherited

To maintain the usefulness of inheritance in a class, the number of inherited methods that are not redefined (overridden) should be relatively greater than the redefined ones.

The *NMI* [40] metric is the ratio of the total number of non-redefined methods to the total number of inherited methods in a class. The following formula measures the value of *NMI*.

$$NMI = \frac{NOHO}{HOP} \times 100 \tag{12}$$

Where:

- NOHO is the number of non-redefined methods in a class.
- HOP is the number of inherited methods in a class.

The ratio of inherited methods should be high. This metric is the opposite of the previously presented *NMO* metric. A low number of inherited methods indicates a lack of specialization. An ideal value of 100% is hardly achievable due to the fact that some behaviors need to be modified in order to satisfy some new requirements.

5.4.14 Specialization Index

Excessive method overriding is undesirable due to the increase in the model complexity and maintenance and hinders reusability. Additionally, an overridden method is presented in a deeper level in the inheritance hierarchy. To that effect, the *NMO* metric is multiplied by the *DIT* metric.

To measure the specialization index for a class in the model, the product of the *NMO* and *DIT* metrics is divided by the total number of methods in the class. Thus, the *SIX* metric is computed using the following formula.

$$SIX = \frac{NMO \times DIT}{NM} \times 100 \tag{13}$$

Where:

- *NMO* is the number of overloaded methods.
- *DIT* is the depth of inheritance value.
- *NM* is the total number of methods in a class.

The deeper in the inheritance hierarchy a class is, the more difficult it would be to efficiently and meaningfully use method overriding. This is due to the fact that it would be more difficult to understand the relationship between the class and its ancestors. In this manner, overridden methods in lower levels of the hierarchy are more easily developed and maintained. A value falling between

0% and 120% is considered in the nominal range. For a root class, the specialization indicator is zero.

5.4.15 Public Methods Ratio

The *PMR* [24] metric measures the access control restrictiveness of a class and indicates how many methods in a class are accessible from other classes. The usefulness of this metric is based on the following considerations.

- Too many public methods defeat the goal of encapsulation which is a desired property of an object oriented design.
- The absence of public methods indicates an isolated entity in the design.

This metric is the ratio of public methods (defined and inherited) to the total number of methods (defined and inherited) in the class. *PMR* is calculated using the following formula.

$$PMR = \frac{PM + PIM}{DM + IM} \tag{14}$$

Where:

- *PM* is the total number of public defined methods in a class.
- *PIM* is the total number of public inherited methods in a class.
- *DM* is the total number of defined methods in a class.
- *IM* is the total number of inherited methods in a class.

A *PMR* nominal range falls between 5% and 50% and indicates that the class has a reasonable number of public methods. A value below 5% is acceptable only for abstract classes otherwise the class functionality will be concealed. Conversely, a value above 50% indicates a lack of encapsulation. Generally, only methods that export some functionality should be visible to other classes.

5.5 Proposed Metrics

In this section, we propose three new metrics for the class diagram that can be used to measure several quality attributes related to complexity, maintainability, accessibility and functionality. This is a continuation of our previous efforts [2] where we investigated the usefulness of several metrics related to class and package diagrams.

The first metric relates to the number of defined attributes in a class and the number of public methods defined. This metric, called the Defined Attributes to Public Method Ratio (DAPMR), is calculated using the following formula:

$$DAPMR = \frac{NDA}{PDM} \tag{15}$$

Where:

- NDA: Number of defined attributes in a class.
- PDM: Number of public defined methods in a class.

Smaller values for this metric indicate an appropriate complexity level of the class methods. This is due to the fact that in general, we need accessor methods for many of the defined (not inherited) attributes, especially when dealing with component-based development. High DAPMR values indicate that some methods might be decomposed into smaller methods in order to provide better maintainability and accessibility. Additionally, we assume that the methods of the derived class are not meant to use the attributes of the base class directly as this would indicate a misplacement of the methods. This assumption is required due to the fact that the related information is not captured in the class diagram notation.

Whereas the first metric can be applied on both specialized and non-specialized classes, our second metric is targeting specialized classes that benefit from inheritance. To that effect, we are interested in what way and to which degree is the subclass specializing the functionality. Thus, we call this metric the Functionality Specialization Degree (FSD). The formula for this metric is as follows:

$$FSD = \frac{NDM + NIM}{NIM} - \frac{NDA + NIA}{NIA} \tag{16}$$

Where:

- NDM: Number of defined methods in a class.
- NIM: Number of inherited methods in a class.
- NDA: Number of defined attributes in a class.
- NIA: Number of inherited attributes in a class.

The FSD metric should be understood as the accompanying added functionality (defined methods) when compared to the added number of attributes. In other words, we can view it as measuring the functionality degree with respect to the added complexity level. A positive value for this metric indicates an appropriate specialization degree. The reason behind it is that an increased number of attributes with little or no functionality added can hardly be beneficial and does not justify the creation of the specialized class. It must be noted that this metric cannot be defined when there is no method or attribute to be inherited.

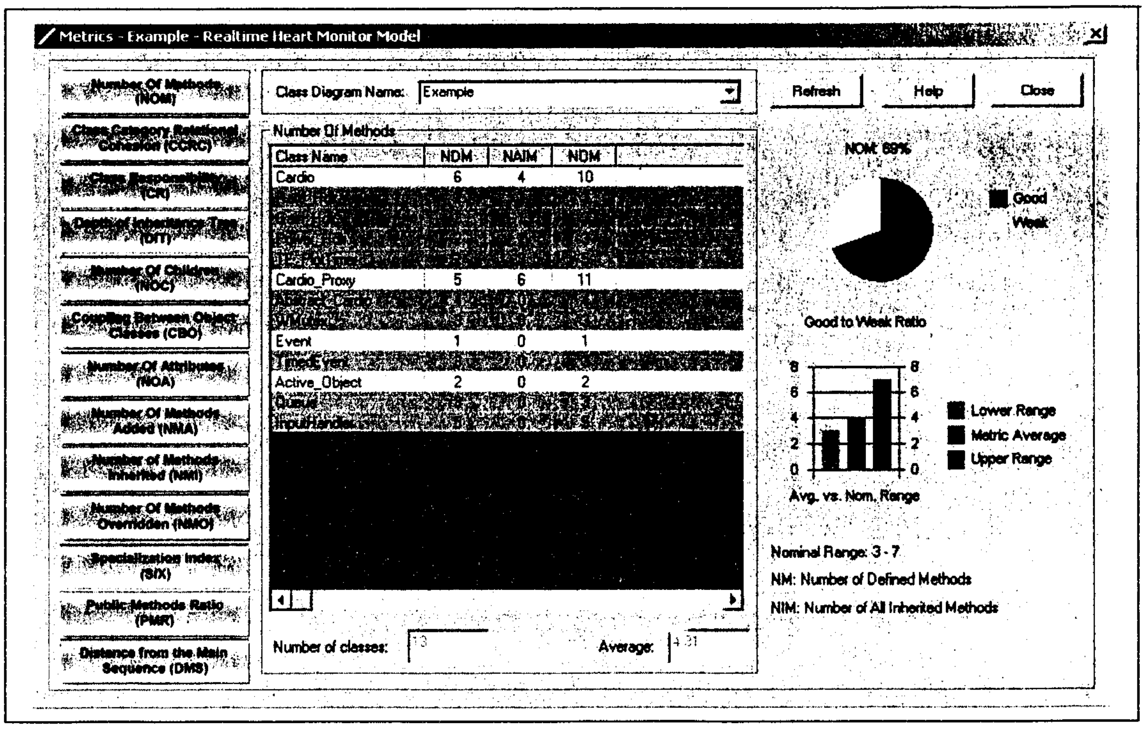
Finally, we present the Class Exposure Ratio (CXR) metric. CXR consists of the ratio between the number of defined private attributes and the number of defined public methods. While for various components it is useful to have *get/set* accessor methods, on the average the exposure degree of the system attributes should be minimal. This is known to be a good practice since it restricts the user from inadvertently altering certain system parameters that might have an adverse side effect. This metric is measured on each single class and averaged for the whole system at the end. For each of the defined private attributes, each method has that attribute as its single parameter can be considered as an accessor method. This metric is measured using the following formula:

$$CXR = \frac{DPrA}{\sum_{a \in DPrA} M(a)} \tag{17}$$

Where:

- DPrA: Defined Private Attributes in a class.
- $M(a)$: Any defined public method that has a as its only argument.

A class that has a CXR value of 1 is highly exposed. If on the average the exposure is high then the design might be reconsidered in order to limit the exposure degree. On the other hand, it is sometimes useful to have an effect (e.g. keeping track of the change, debugging, etc.) based on setting a certain system parameter through an accessor method instead of allowing that particular attribute to be defined as publicly accessible.


Metrics - Example - Realtime Heart Monitor Model
Number Of Methods (NOM)
Class Category Relational Cohesion (CCRC)
Class Responsibility (CR)
Depth of Inheritance Tree (DIT)
Number Of Children (NOC)
Coupling Between Object Classes (CBO)
Number Of Attributes (NOA)
Number Of Methods Added (NMA)
Number of Methods Inherited (NMI)
Number Of Methods Overridden (NMO)
Specialization Index (SIX)
Public Methods Ratio (PMR)
Distance from the Main Sequence (DMS)
Refresh
Help
Close
NOM: 69%
Good
Weak
Good to Weak Ratio
Lower Range
Metric Average
Upper Range
Avg. vs. Nom. Range

Class Diagram Name: Example

Number Of Methods

Class Name	NDM	NAIM	NDM
Cardio	6	4	10
[illegible]	[illegible]	[illegible]	[illegible]
Cardio_Proxy	5	6	11
[illegible]	[illegible]	[illegible]	[illegible]
Event	1	0	1
[illegible]	[illegible]	[illegible]	[illegible]
Active_Object	2	0	2
[illegible]	[illegible]	[illegible]	[illegible]

Number of classes: 13

Average: 4.31

Nominal Range: 3 - 7

NM: Number of Defined Methods

NIM: Number of All Inherited Methods

Figure 5.1: Snapshot of the Metrics Tool

5.6 Object Oriented Metrics Tool

In this section, we give an overview of our software metrics tool. As mentioned in Chapter 1, we used Artisan Real-time Studio [59], a modeling tool that supports both UML 2.0 and SysML artifacts. The metamodel of Artisan Real-time Studio is based on an object oriented database system. The metrics tool is developed using Microsoft Visual C++ 6.0 and accesses Artisan data-store through a provided ActiveX component. A snapshot of the metrics tool is shown in Figure 5.1.

The tool implements the aforementioned fifteen metrics. The toolkit is an automated application with GUI frontend that enables the UML designer to directly measure the quality of class and package diagrams. Moreover, the toolkit reflects the modified changes of the diagram into the verification tool automatically. Furthermore, for each targeted metric, the tool shows the ratio of the classes that satisfy the nominal range in the diagram to the total number of classes in the diagram. The toolkit provides as well a feedback to the designer on the assessed models and some hints that may be used to enhance the design. The results of our metrics tool along with the accompanying example were presented in [2,4].

5.7 Case Study

We selected an example depicting a real-time heart monitoring system. The diagram consists of three packages. The first one contains the windows components that display the monitoring results. The second package contains the platform specific heart monitoring tools, whereas, the third package contains the heart monitoring components.

This diagram is a good example to be tested due to the different types of relationships among the classes. Our tool implements a set of fifteen metrics [24] for class and package diagram. In the following paragraphs we briefly present our assessment results. When applying these metrics on the diagram in Figure 5.2, the analysis results indicate that some classes in the model are complex thus having a weak reusability potential.

Table 5.1 shows the metrics related to package diagrams. The distance from the main sequence metric (DMS) measures the balance between the abstraction and instability levels in the package. As shown in the table, the three packages in the diagram fall within the nominal range of DMS.

Since the *Abstraction* and *Instability* metrics do not have nominal ranges due to the difference

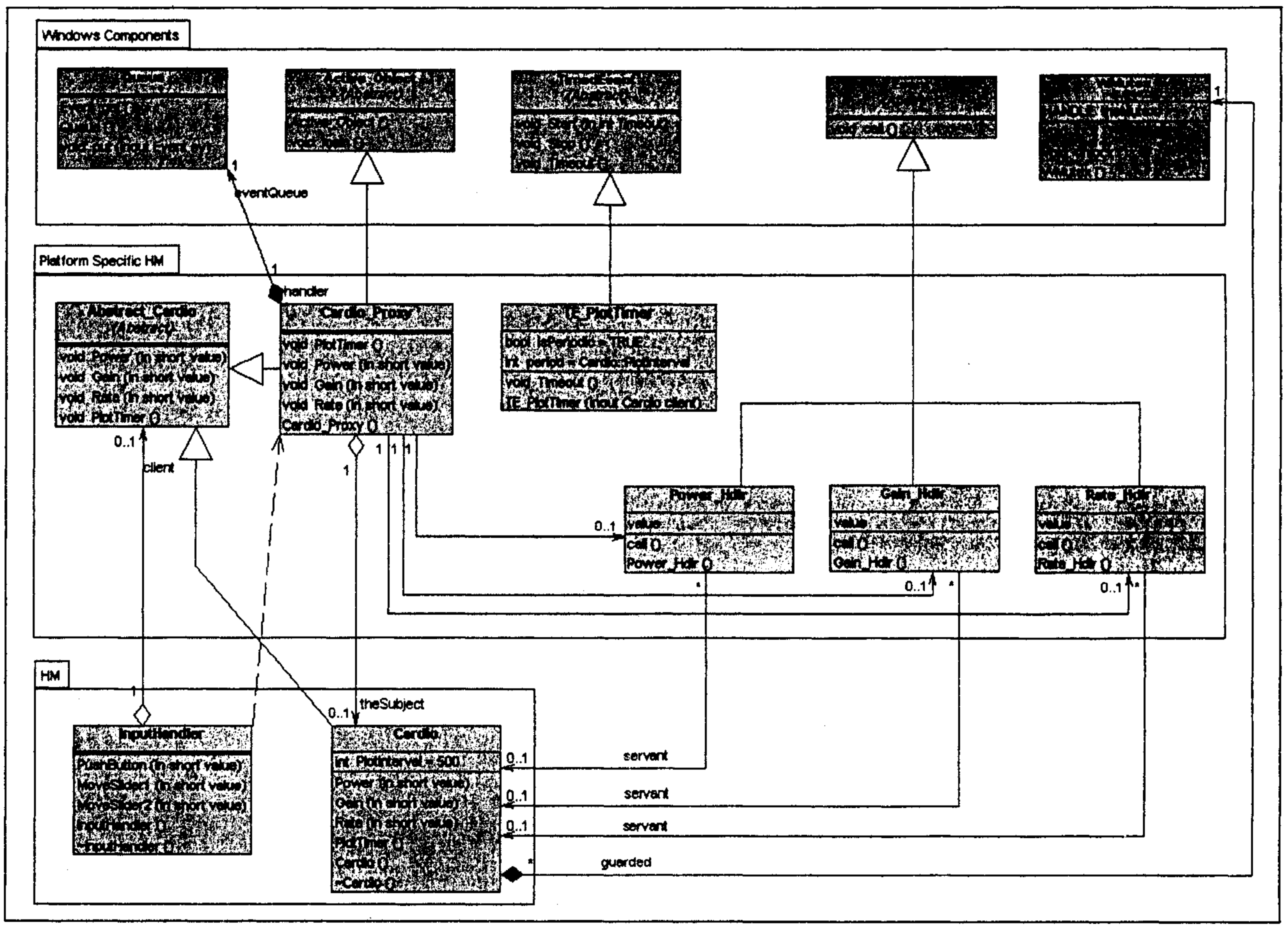


Figure 5.2: Class and Package Diagrams Example

in the design perspectives, the DMS is a compromise between their values. *Abstraction* and *Instability* metrics do not have nominal ranges due to the fact that packages should depend on other packages in order to employ compositionality. However, they should also be easily modifiable. In the table above, the zero abstractness value for the three packages shows that these packages are not easily extendable and modifiable. Also, the metric in the second column shows a relatively high instability value indicating that the three packages are subject to change if other packages do change.

Package Name	A	I	DMS
Platform Specific HM	0	49	51
HM	0	49	51
Windows Components	0	50	50
Average	0	50	50
Nominal Range	-	-	50 - 100%

Table 5.1: Package Diagram Metrics

Table 5.2 presents the analysis results of the class diagram inheritance related metrics. The (Depth of Inheritance Tree) DIT metric shows a proper use of inheritance. Moreover, the use of inheritance in this diagram does not have a negative impact on the its complexity level. Furthermore, our tool results show that the diagram has a shallow inheritance tree which indicates a good level of understandability and testability. With respect to the Number Of Children (NOC), the analysis shows that only four classes in the diagram have a good NOC value. In a class diagram, the number of children is an indication of how a class is being reused in the diagram.

The analysis results also show that five classes in the diagram have weak (Number Of Methods) NOM value. On the other hand, the class diagram has an overall NOM that lies in the nominal range. The problem of unsuitable NOM values may be solved by modifying the class diagram by decomposing the existing classes to smaller new classes to share the number of methods that exceed the nominal range. Consequently, classes in the class diagram will be more reusable.

Table 5.2 shows only one class in the Number Of Attributes (NOA) nominal range. This allows for further enhancement by adding new attributes to the non-abstract classes in the diagram. Moreover, a class with a high number of attributes increases its size.

The Number of Methods Added (NMA) measures the inheritance usefulness degree. Three

Class Name	DIT	NOC	NOM	NOA	NMA	NMI	NMO	SIX
Cardio	1	0	10	1	6	0	4	40
Rate_Hdlr	1	0	3	1	2	0	1	33
Gain_Hdlr	1	0	3	1	2	0	1	33
Power_Hdlr	1	0	3	1	2	0	1	33
TE_PlotTimer	1	0	5	2	2	67	3	43
Cardio_Proxy	1	0	11	0	5	33	4	0
Abstract_Cardio	0	2	4	0	4	0	0	0
WMutex	0	0	3	1	3	0	0	0
Event	0	3	1	0	1	0	0	0
TimedEvent	0	1	3	0	3	0	0	0
Active_Object	0	1	2	0	2	0	0	0
Queue	0	0	3	0	3	0	0	0
Input_Handler	0	0	5	0	5	0	0	0
Average	0.46	0.88	4.31	0.54	3.08	25	1.08	18.38
Nominal Range	1 - 4	1 - 4	3 - 7	2 - 5	0 - 4	50 - 100%	0 - 5	0 - 120%

Table 5.2: Class Diagram Inheritance Related Metrics

classes have a high NMA value indicating a misuse of inheritance. Classes with high NMA may be difficult to reuse, whereas, classes with no specialization and having large number of methods may impede other classes from reusing their functionality requiring the decomposition into smaller specialized classes in order to improve the design.

Table 5.2 shows a single class in the inheritance hierarchy satisfying the Number of Methods Inherited (NMI) nominal range. Concerning the Number of Methods Overridden (NMO) metric, our analysis shows that all the classes in the diagram fall in the nominal range.

The last metric in Table 5.2 shows that all the classes in the diagram comply to the nominal range of the specialization index (SIX). The latter reflects the overall performance of the class diagram from the perspective of inheritance in object oriented design.

The Coupling Between Object (CBO) classes metric measures the level of coupling between classes, denoting an increase in the complexity for high coupling. Table 5.3 shows seven classes

Class Name	CR	CCRC	CBO	PMR
Cardio	0	200	1	100
Rate_Hdlr	0	200	1	100
Gain_Hdlr	0	200	1	100
Power_Hdlr	0	200	1	100
TE_PlotTimer	0	100	0	100
Cardio_Proxy	0	700	5	100
Abstract_Cardio	0	100	1	100
WMutex	0	0	0	100
Event	0	0	0	100
TimedEvent	0	0	0	100
Active_Object	0	0	0	100
Queue	0	0	0	100
Input_Handler	0	200	2	100
Average	0	146	0.92	100
Nominal Range	20 - 75%	150 - 350%	1 - 4	5 - 50%

Table 5.3: Class Diagram General Metrics

outside the CBO nominal range while six classes are falling within it. This shows an increased complexity and suggests further modification by reducing the number of relationships between the classes.

The Class Category Relational Cohesion (CCRC) measures the cohesion of classes within the diagram. This metric reflects the diagram's architecture strength. Table 5.3 shows a good CCRC level for only five classes, whereas the remaining eight classes have a weak CCRC level. We can also see that the average CCRC is outside the nominal range indicating a lack of cohesion between the classes.

The Class Responsibility (CR) results in Table 5.3 show that none of the classes in the diagram is implementing pre-conditions and/or post-conditions. CR is measured in the cases when a class method should be responsible to check whether a message is correct before taking any action. In the current example, the CR value can be enhanced by adding pre/post-conditions to the methods that need to check the validity of messages prior or after the execution of an action. In the design of a class diagram, the use of pre/post-conditions should be carefully considered. Therefore, this metric is useful to check systems with real-time messaging.

Finally, for the PMR metric, Table 5.3 shows that all methods in the class diagram are accessible which inhibits encapsulation in the diagram. This requires some adjustment of the access control level for all the classes in the diagram.

5.8 Summary

In this chapter we explained our adopted set of metrics. We have demonstrated using a case study the usefulness of these metrics in assessing the quality of software systems. Our case study shows how applying different object oriented techniques can affect different quality attributes such as reusability and complexity. Moreover, we provided an analysis for each metric in our tool.

Chapter 6

Conclusion

Increased complexity in emerging systems required the existence of a discipline that develops and exploits structured, efficient approaches to analysis and design to solve complex engineering problems. Systems engineering is a discipline that aims to successful realization of complex systems. The increased difficulties in coping with systems design and maintenance led to the emergence of systems modeling languages such as UML 2.0 and SysML. INCOSE and OMG collaborated in the development of the aforementioned modeling languages in order to capture all the systems engineering aspects and properties.

In this work, we targeted some techniques aimed to automate verification and validation of software and systems engineering designs. To that effect, software engineering techniques and formal techniques like model-checking can be useful in building automatic tools that assist verification and validation of systems design. The benefit of applying early verification and validation at the design stages has the potential to eliminate a huge cost of correcting errors and maintaining systems in the subsequent phases of the system development.

The outcome of this research work demonstrated a unified paradigm that can be used for the automated assessment of software and systems engineering design models focusing on both structural and behavioral aspects captured by UML 2.0 class diagrams and UML 2.0 activity diagrams respectively. The latter, captures important areas such business processes, workflow systems modeling and real-time systems. In Chapter 2, we presented the state-of-the-art in the verification and validation research initiatives targeting UML structural and behavioral diagrams. Moreover, Chapter 3 outlined the historical context and background that led to the emergence of the UML 2.0 and SysML modeling languages.

Furthermore, Chapter 4 presented the informal syntax and semantics of UML 2.0 activity diagram focusing on its control flow aspect. In the same chapter, the NuSMV model checker was introduced along with its potential for verifying behavioral descriptions such as the activity diagrams. An illustrative business process case study assessment was elaborated detailing the phases required for the verification and validation of activity diagrams. Also, a discussion about the model checking feasibility and the related open problems concluded the chapter.

Moreover, Chapter 5 demonstrated the usefulness of software engineering metrics in the assessment of structural aspects of a system captured by UML class diagrams. To that effect, a set of fifteen was discussed in the context of a relevant example. A side benefit of this research was the realization of a software package dedicated to the assessment of UML class diagrams quality attributes.

The future work consists of developing new object oriented metrics that can be applied for the assessment of class diagrams and extending the metrics concept to cover the behavioral diagram semantic model in a synergetic way. Similarly, a future research direction consists in using program analysis techniques such as data and control flow analysis in order to slice the transition system corresponding to UML behavioral diagrams such as activity diagrams. This has the potential to leverage the effectiveness of the model checking procedure.

Finally, the demonstrated empirical results related to the assessment of the activity diagram might be augmented with a theoretical base that would strengthen the confidence in the proposed method. This may consist in a formal syntax and operational semantics for expressing activity diagrams structure and behavior.

Bibliography

[1] National Aeronautics and Space Administration (NASA). Software Quality Metrics for Object Oriented System Environments. Technical Report SATC-TR-95-1001, National Aeronautics and Space Administration, Goddard Space Flight Center, Greenbelt Maryland 20771, JUNE 1995.

[2] L. Alawneh, M. Debbabi, Y. Jarraya, A. Soeanu, and F. Hassaine. A unified approach for verification and validation of systems and software engineering models. In *13th Annual IEEE International Symposium and Workshop on Engineering of Computer Based Systems (ECBS'06)*, pages 409–418, 2006.

[3] Luay Alawneh, Mourad Debbabi, Fawzi Hassaïne, Yosr Jarraya, Payam Shahi, and Andrei Soeanu. Towards a unified paradigm for verification and validation of systems engineering design models. In *IASTED Conf. on Software Engineering*, pages 282–287, 2006.

[4] Luay Alawneh, Mourad Debbabi, Fawzi Hassaïne, and Andrei Soeanu. On the verification and validation of uml structural and behavioral diagrams. In *IASTED Conf. on Advances In Computer Science and Technology - 2006*, 2006.

[5] Inc. Averant. Static Functional Verification with Solidify, a New Low-Risk Methodology for Faster Debug of ASICs and Programmable Parts. Technical report, Averant, Inc.s, 2001.

[6] Jagdish Bansiya and Carl G. Davis. A hierarchical model for object-oriented design quality assessment. *IEEE Trans. Softw. Eng.*, 28(1):4–17, 2002.

[7] J. Paulo Barros and Jens B. Jorgensen. Model transformations for an elevator controller: Coloured petri nets in object-oriented analysis and design. In *Second International Workshop on Model-Based Methodologies for Pervasive and Embedded Software (MOMPES 2005), Renes, France.*, June, 2005.

[8] Shoham Ben-David, Tamir Heyman, Orna Grumberg, and Assaf Schuster. Scalable distributed on-the-fly symbolic model checking. In *Formal Methods in Computer-Aided Design*, pages 390–404, 2000.

[9] Barry W. Boehm and Victor R. Basili. Software Defect Reduction Top 10 List. *IEEE Computer*, 34(1):135–137, 2001.

[10] G. Booch. *Object-Oriented Analysis and Design with Applications.* Addison-Wesley, second edition, 1997.

[11] Universitat Bremen. udraw(graph)tool. `http://www.informatik.uni-bremen.de/uDrawGraph/en/index.html`.

[12] Lionel C. Briand, Premkumar T. Devanbu, and Walcelio L. Melo. An investigation into coupling measures for c++. In *International Conference on Software Engineering*, pages 412–421, 1997.

[13] F. Brito, e Abreu, and W. Melo. Evaluating the impact of object-oriented design on software quality, 1996.

[14] S. R. Chidamber and C. F. Kemerer. A Metrics Suite for Object Oriented Design. *IEEE Trans. Softw. Eng.*, 20(6):476–493, 1994.

[15] Alessandro Cimatti, Edmund M. Clarke, Fausto Giunchiglia, and Marco Roveri. Nusmv: A New Symbolic Model Verifier. In *cav*, 1999.

[16] Communications Committee. The International Council on Systems Engineering (incose). `http://www.incose.org/practice/whatissystemseng.aspx`.

[17] Pallab Dasgupta, Arindam Chakrabarti, and P. P. Chakrabarti. Open computation tree logic for formal verification of modules. In *ASP-DAC '02: Proceedings of the 2002 conference on Asia South Pacific design automation/VLSI Design*, page 735, Washington, DC, USA, 2002. IEEE Computer Society.

[18] Alexandre David, Gerd Behrmann, Kim G. Larsen, and Wang Yi. A Tool Architecture for the Next Generation of Uppaal. Technical report, 2002.

[19] Semantik der UML 2.0. http://www4.in.tum.de/lehre/seminare/hs/ws0405/uml/20040720.pdf.

[20] Gregor Engels, Jochen M. Kuster, Reiko Heckel, and Marc Lohmann. Model-Based Verification and Validation of Properties. In Roswitha Bardohl and Hartmut Ehrig, editors, *Electronic Notes in Theoretical Computer Science*, volume 82. Elsevier, 2003.

[21] Rik Eshuis and Roel Wieringa. Tool support for verifying uml activity diagrams. *IEEE Transactions on Software Engineering*, 30(7):437–447, 2004.

[22] Object Management Group. http://www.omg.org.

[23] Object Management Group. Uml for Systems Engineering, 2003.

[24] USAF Research Group. Object Oriented Model Metrics. Technical report, The United States Air Force Space and Warning Product-Line Systems, http://www.cin.ufpe.br/ inspector/relacionados/Object-oriente Model Metrics Document.htm, 1996.

[25] Esther Guerra and Juan de Lara. A framework for the verification of uml models. examples using petri nets. In *JISBD*, pages 325–334, 2003.

[26] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[27] Gerard J. Holzmann. The Model Checker SPIN. *Software Engineering*, 23(5):279–295, 1997.

[28] Lin Hsin-Hung. A Research of Model Checking UML Statechart Diagrams. Master's thesis, Japan Advanced Institute of Science and Technology, 2003.

[29] Hugo/RT. http://www.pst.ifi.lmu.de/projekte/hugo/.

[30] IEEE. *IEEE Std 610.12-1990, IEEE Standard for Software Verification and Validation*, 1990.

[31] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. In *Proceedings of the Fifth Annual IEEE Symposium on Logic in Computer Science*, pages 1–33, Washington, D.C., 1990. IEEE Computer Society Press.

[32] K.L. McMillan. The SMV system. Technical Report CMU-CS-92-131, 1992.

[33] Alexander Knapp, Stephan Merz, and Christopher Rauh. Model Checking Timed UML State Machines and Collaborations. Technical report, Institut fur Informatik, Ludwig-Maximilians-Universitat Munchen and Institut fur Informatik, Technische Universitat Munchen, 2002.

[34] D. Kroening. Application Specific Higher Order Logic Theorem Proving, 2002.

[35] Unified Modeling Language. http://www.uml.org/.

[36] Diego Latella, Istvan Majzik, and Mieke Massink. Towards a formal operational semantics of uml statechart diagrams. In *Proceedings of the IFIP TC6/WG6.1 Third International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS)*, page 465, Deventer, The Netherlands, The Netherlands, 1999. Kluwer, B.V.

[37] W. Li and S. Henry. Maintenance metrics for the object oriented paradigm. In *First International Software Metrics Symp.*, pages 52–60, 1993.

[38] Wei Li and Sallie Henry. Object-oriented metrics that predict maintainability. *J. Syst. Softw.*, 23(2):111–122, 1993.

[39] Johan Lilius and Ivan Porres Paltor. vUML: a tool for verifying UML models. Technical Report TUCS-TR-272, 18, 1999.

[40] Mark Lorenz and Jeff Kidd. *Object-Oriented Software Metrics: a Practical Guide*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1994.

[41] Formal Systems Europe (Ltd). *Failures-Divergence-Refinement: FDR2 User Manual*. 1997.

[42] Robert C. Martin. OO Design Quality Metrics. 1994.

[43] McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, 2:308–320, 1976.

[44] John McLeod. Advances in simulation. *Advances in Computers*, 9:23–49, 1968.

[45] Stephan Merz. Model Checking: A Tutorial Overview. In F. Cassez et al., editor, *Modeling and Verification of Parallel Processes*, volume 2067 of *Lecture Notes in Computer Science*, pages 3–38. Springer-Verlag, Heidelberg, 2001.

[46] J. Miller and J. (eds.) Mukerji. MDA Guide Version 1.0. *OMG Document.*, May 2003. omg/2003-05-01.

[47] Object Management Group (OMG). Meta-Object Facility (MOF) Specification, 2002.

[48] Object Management Group (OMG). UML 2.0 Superstructure Specification, 2003.

[49] Object Management Group (OMG). XML Metadata Interchange (XMI) Specification, 2003.

[50] What Is OMG-UML and Why Is It Important? http://www.omg.org/news/pr97/umlprimer.html.

[51] International Standard Organisation. http://www.iso.org.

[52] SysML Partners. System Modeling Language: SysML, 2004.

[53] Doron Peled. Combining partial order reductions with on-the-fly model-checking. In *CAV '94: Proceedings of the 6th International Conference on Computer Aided Verification*, pages 377–390, London, UK, 1994. Springer-Verlag.

[54] Tom Pender. *UML Bible*. Wiley, 2003.

[55] Marco Roveri. PSL Sugar: Formal Specification Language. Lecture for Course Advanced Model Checking, September 2004.

[56] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language Reference Manual Second Edition*. Addison-Wesley, 2005.

[57] Assaf Schuster. Scalable distributed model checking: Experiences, lessons, and expectations. *Electr. Notes Theor. Comput. Sci.*, 89(1), 2003.

[58] ARTiSAN Software. ARTiSAN Real-time Modeler. `http://www.microprocess.com/agls/documents/ARTISAN/FichesProduits/Model_4.pdf`. Datasheet.

[59] ARTiSAN Software. ARTiSAN Real-time Studio. `http://www.artisansw.com/pdflibrary/Rts_5.0_datasheet.pdf`. Datasheet.

[60] G.C. Tugwell, J.D. Holt, C.J. Neill, and C.P. Jobling. Metrics for Full Systems Engineering Lifecycle Activities (MeFuSELA). In *Proceedings of the Ninth International Symposium of the International Council on Systems Engineering (INCOSE 99)*, Brighton, U.K., 1999.

[61] R. W. Whitty. Software testing techniques, by boris beizer, van nostrand reinhold, second edition, 1990 and testing computer software, by c. kaner, j. falik and h. q. nguyen, van nostrand reinhold, second edition, 1993 (book review). *Softw. Test., Verif. Reliab.*, 2(4):215–216, 1992.

[62] Lei Zhen and Guangzhen Shao. Analysis patterns for oil refineries.